VHDL IMPLEMENTATION OF 32-BIT INTERLOCK COLLAPSING ALU

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

in

Electrical Engineering

By

HIMANSHU SHEKHAR ACHARYA
BIBHUTI PRASAD SAHOO
NEM KUMAR NEERAJ

Department of Electrical Engineering

National Institute of Technology

Rourkela

2007

VHDL IMPLEMENTATION OF 32-BIT INTERLOCK COLLAPSING ALU

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
in
Electrical Engineering

By

HIMANSHU SHEKHAR ACHARYA
BIBHUTI PRASAD SAHOO
NEM KUMAR NEERAJ

Under the Guidance of

Prof. S. Das

Department of Electrical Engineering

National Institute of Technology

Rourkela

2007

CERTIFICATE

This is to certify that the thesis entitled,"VHDL implementation of 32 bit Interlock Collapsing ALU" submitted by Sri Himanshu Shekhar Acharya, Nem kumar Neeraj, Bibhuti Prasad Sahoo in partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Electrical Engineering at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Date :

Place:

Prof. S. Das
Dept. of Electrical Engg.
National Institute of technology
Rourkela-769008

ACKNOWLEDGEMENT

I would like to articulate my deep gratitude to my project guide Prof. S. Das ; Department of Electrical Engg , NIT Rourkela who has always been my motivation for carrying out the project.

I wish to extend my sincere thanks to Prof. P. K. Nanda, Head of our Department, for approving the request for the financial aid to develop the model.

It is my pleasure to refer Xilinx Project Navigator for simulating & Microsoft Word exclusive of which the compilation of this report would have been impossible.

An assemblage of this nature could never have been attempted with our reference to and inspiration from the works of others whose details are mentioned in references section. I acknowledge my indebtedness to all of them. Last but not the least, my sincere thanks to Prof D.Patra , Mr Chandu (M-tech senior) and all of my friends who have patiently extended all sorts of help for accomplishing this undertaking.

HIMANSHU SHEKHAR ACHARYA(10302007)

BIBHUTI PRASAD SAHOO(10402068D)

NEM KUMAR NEERAJ(10302053)

TABLE OF CONTENTS

Page

ABSTRACT
v

CHAPTER 1 INTRODUCTION----------
1

CHAPTER 2 THEORY ----
3

2.1 Introduction---------- 4

2.2 Preliminary Design issues of ICALU---------- 19

2.3 Working of the ICALU model---------- 24

2.4 Design and Implementation of the ICALU---------- 25

CHAPTER 3 PRELIMINARY DESIGN ISSUES OF ICALU --- 19

4.1 Design of CLA input stage---------- 26

4.2 Design of PRE-CLA logic block---------- 29

4.3 Binary adders and arithmetic ---------- 33

4.4 Design of CSA stage---------- 40

4.5 Design of CLA stage---------- 42

4.6 Design of POST-CLA logic block---------- 46

CHAPTER 4 ICALU DATAFLOW MODEL --- 48

5..1 Reduced ICALU model---------- 50

5..2 ALU1---------- 52

5.3 Interlock collapsing unit---------- 53

5.4 Estimation of relative delay between ALU1 and ICALU-- 53

5.5 Determination of instruction cycle lengths of a machine

with and without ICALU---------- 55

CHAPTER 5 PERFORMANCE ANALYSIS---------- 58

CHAPTER 6 TESTING PROCEDURES ---------- 63

CHAPTER 7 SIMULATION RESULTS---------- 67

CHAPTER8 CONCLUSIONS---------- 73

CHAPTER 9 REFERENCE---------- 75

VHDL SOURCE CODE---------- 77

DETERMINATION OF INSTRUCTION LENGTHS

FOR FREQUENTLY EXECUTED INSTRUCTIONS

ABSTRACT

An important area in computer architecture is parallel processing. Machines employing parallel processing are called parallel machines. A parallel machine executes multiple instructions in one cycle. However, parallel machines have a limitation, they cannot execute interlocked instructions. They are executed in serial like any serial machine. It takes more than one cycle to execute multiple instructions causing performance degradation. In addition there is hardware underutilization as a result of serial execution in parallel machine.

The solution requires a special kind of device called “Interlock collapsing ALU”. The Interlock Collapsing ALU, unlike conventional 2-1 ALU’s is a 3-1 ALU. The proposed device executes the interlocked instructions in a single instruction cycle, unlike other parallel machines, resulting in high performance. The resulting implementation demonstrates that the proposed 3-1 Interlock Collapsing ALU can be designed to outperform existing schemes for ICALU, by a factor of at least two. The ICALU is implemented in VHDL. Its functionality is verified through simulation.

Chapter 1

INTRODUCTION

INTRODUCTION:

BACKGROUND:

Parallel machines cannot execute interlocked instruction concurrently.Interlocked instructions or instruction with dependencies cannot be executed concurrently in a parallel machine, thus degrading the performance of the machine. The thesis investigates a solution, called, “interlock collapsing”, to execute these interlocks concurrently. The solution requires a special kind of a device called the Interlock collapsing ALU. The Interlock collapsing ALU, unlike conventional 2-1 ALU’s, is a 3-1 ALU.

The proposed ALU, in addition to collapsing these interlocks also should be implemented in identical stages as the conventional ALU’s. A functional model of the ICALU is assumed initially. The functional model is optimized by optimizing the model’s individual blocks. The design and optimization of each block is discussed in separate chapters.

Finally, two parallel machines with and without the ICALU are compared with regard to their execution times. The effect of variation of percentage interlocks in a given code on the execution times and the percentage speed ratio of the parallel machines is studied.

The ICALU is implemented in VHDL. Its functionality is verified through simulation.

Chapter 2

THEORY

Preliminary design issues of ICALU

Basic computer architecture

Instruction formats

Parallel machines

Interlocked instructions

ICALU

Why VHDL

2.1 INTRODUCTION ::

Computers have markedly changed over the last decade. Features, performance, and memory sizes representing a computer that filled a room with equipment and cost millions of dollars a decade ago now sit on top of a desk. High performance computers are increasingly in demand in the areas of industrial automation, medical diagnosis, aerodynamics simulation, military defense, signal processing, artificial intelligence, expert systems and socioeconomic, among many other scientific and engineering applications. This revolution has been brought about by major improvements in computer architecture and processing techniques and the enabling technology of Very Large Scale Integration (VLSI).

This thesis involved developing a novel technique to speed up instruction execution in parallel computers. Before moving any further, an overview of basic computer components and its many related terms is necessary.

2.1.1 COMPUTER ARCHITECTURE ::

WHAT IS COMPUTER ARCHITECTURE ?

Computer architecture involves the design of various aspects of computer design such as memory design, bus structure, internal Central Processing Unit, instruction set and the hardware implementation of the machine.

The aim of a computer architect is to design a computer that meets the functional requirements as well as price and performance goals.

2.1.2 BASIC COMPUTER COMPONENTS ::

Computer architecture has changed incredibly over the years. One element has remained constant throughout the years, and that is the Von Neumann concept of computer design.

Von Neumann architecture is composed of three distinct components(or subsystems) : a central processing unit (CPU), memory and input/output (I/O) interfaces. Fig 2.1 represents one of the several possible ways of connecting these components together.

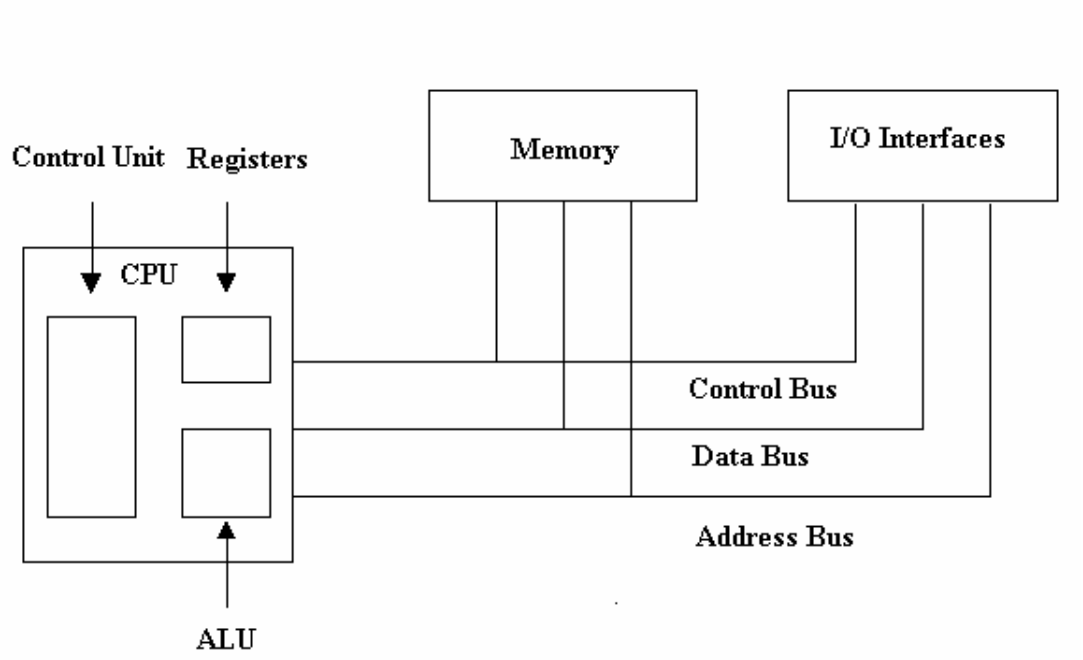


Fig 2.1

(Basic computer components)

1) THE CPU ::

The heart of any computing unit is the Central processing unit. It is responsible for executing instructions in the computer. Fig 2.2 shows the block diagram of a simple

CPU. Usually CPU's are available on single chips and are called microprocessors. The major components of a CPU are ::

I) CONTROL UNIT ::

The Control Unit determines the order in which instructions should be executed. It interprets the machine instructions. The execution of each instruction is determined by a sequence of control signals produced by the control unit. In other words, the control unit governs the flow of information through the system by issuing control signals to different components. For example, to perform an addition operation, it sets the appropriate signals to appropriate components so that an addition operation results.

II) ALU ::

The Arithmetic and Logic Unit (ALU) is arguably the most important part of the CPU. The ALU performs the decision making operations (logical) and arithmetic operations. Arithmetic operations involve functions such as addition, subtraction, multiplication and division. It also performs the basic logic functions such as AND, OR, XOR, and so on. There are a variety of techniques to design these functions. It is most complex with regard to design, amongst all the components of the computer, and it also contributes to most of the delay. Thus, the design of the ALU is critical to the speed of the computer.

III) REGISTER ARRAY ::

The Register Array consists of a number of temporary storage locations or registers. Because the registers are often on the same chip and directly connected to the

control unit, they have faster access than memory. The ALU and the register array are together called as the ‘dataflow’ of the computer.

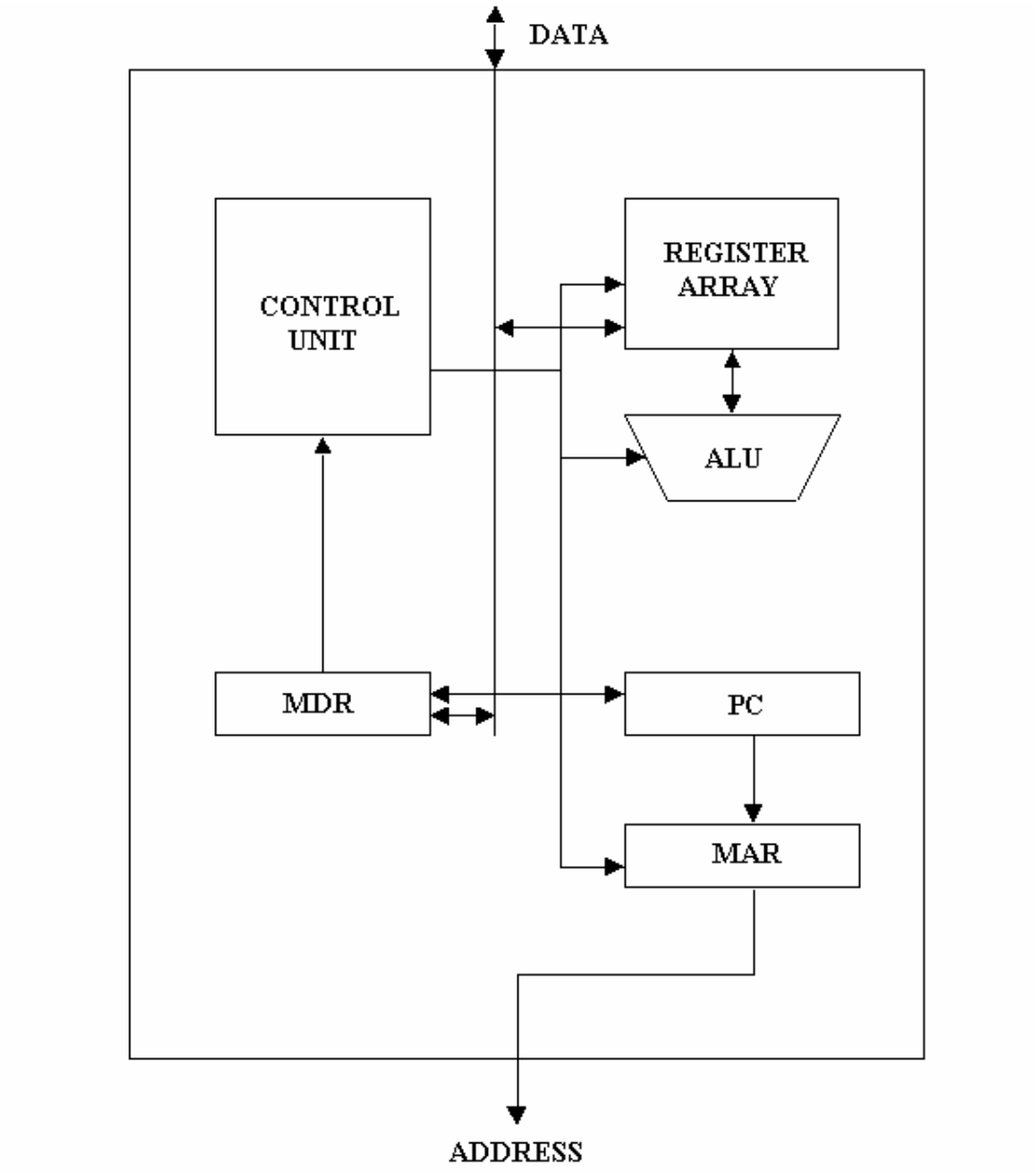


Fig 2.2

(The CPU)

IV) PC ::

An instruction is fetched from the memory by placing the address of the location in the program counter (PC). It keeps the address of the next instruction to be executed.

V) MAR AND MDR ::

The CPU communicates with the memory modules through the Memory. Whenever data or instruction is fetched from memory, it is first placed in the MDR. Address is sent out of the CPU through the MAR.

2) MEMORY ::

The computer's memory is used to store program instructions and data. Two of the commonly used type of memories are, RAM (random-access memory) and ROM (read-only memory). RAM stores the data and general-purpose programs that the machine executes and is temporary. Its contents can be changed any time or can be erased when power to the computer is turned off. ROM is permanent and is used to store the initial boot-up instructions of the machine.

3) INPUT/OUTPUT INTERFACES ::

The I/O interfaces allow the computer to communicate to the user and to secondary storage devices like the disk and tape drives.

2.1.3 INSTRUCTION FORMAT ::

An instruction is a group of binary bits that tell the computer what has to be done. Any computer instruction has two parts ::

i) **Opcode ::** Opcode (stands for operation code) field determines the function of the instruction, i.e. it contains the operation that is to be performed by the CPU.

ii) **Operand ::** Operand is the data on which the intended operation is performed.

Opcode	Operand	……	Operand

Fig 2.3a

(Instruction format)

ADD R_d, R_s

Fig 2.3b

(An instruction)

Fig 2.3a shows the format of an instruction. Fig 2.3b is an example where the opcode is ADD. The operand are the data within the registers, R_d and R_s.

R_d – Destination Register.

R_s – Source Register.

The above instruction does the following operation :

$R_d \leftarrow R_d + R_s$

1) A TYPICAL COMPUTER PROGRAM::

The following assembly language program adds the two numbers ::

MOVI R1, FF -- Load register R1 with number 1.

MOVI R2, OF -- Load register R2 with number 2.

ADD R1, R2 -- Add contents of R1 with that of R2, store Result in R1.

SUBI R1, 10 -- Subtract a number from R1, store result in R1.

2) STEPS IN INSTRUCTION EXECUTION::

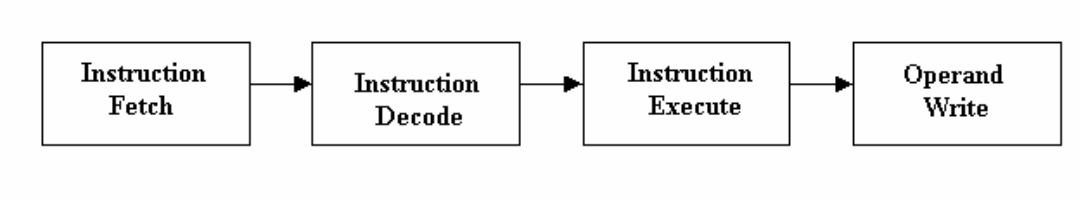


Fig 2.4

(Phases of instruction execution process.)

Fig 2.4 outlines the steps in instruction execution [1,4]. Typically, it consists of four machine cycles. A 'machine cycle' is the time taken to complete one phase of an operation i.e., instruction fetch or instruction decode, etc. An instruction does not necessarily have all the machine cycles. The various phases are explained ::

I) INSTRUCTION FETCH (IF) ::

During this phase the Program Counter loads the MAR with the address of the instruction to be executed. The address is sent out to the memory over the address lines. The instruction is fetched from the memory location and placed in the MDR. This is a 'memory-read' operation.

II) INSTRUCTION DECODE (ID) ::

The Control Unit reads the instruction from the MDR and decodes it. The control unit examines the opcode of the instruction and decides whether data needs to be retrieved from the memory. Once the data is in the CPU, the control unit sets the appropriate signals to perform the required operations, i.e., arithmetic, logic etc.

III) INSTRUCTION EXECUTE (EX) ::

During the instruction execute phase the ALU is loaded with the operands, and it performs the necessary operations as set by the control unit.

IV) OPERAND WRITE OR MEMORY WRITE (MW) ::

` The result of the ALU is placed in the MDR. The PC writes the address of the memory location where the data has to be written into the MAR. A memory write operation is performed by the CPU to transfer the contents of MDR into memory.

An 'instruction cycle' is the time taken by the computer to complete the execution of one instruction, i.e., the sum of all machine cycles.

2.1.4 PARALLEL MACHINES ::

An important area in computer architecture is parallel processing. Machines (computers) employing parallel processing are called parallel machines. A parallel machine executes multiple instructions in parallel, in one cycle, compared to a serial machine (discussed so far) that can execute only one instruction. Thus a parallel machine is faster than a serial machine.

In a parallel machine, a number of execution units (ALU's) are connected in parallel, so that each unit is able to handle an instruction. But for practical reasons the number is limited to two. For example, if two such units are present in the processor, two instructions can be handled concurrently resulting in faster execution. Fig 2.5 shows a simple block diagram of a parallel machine unit.

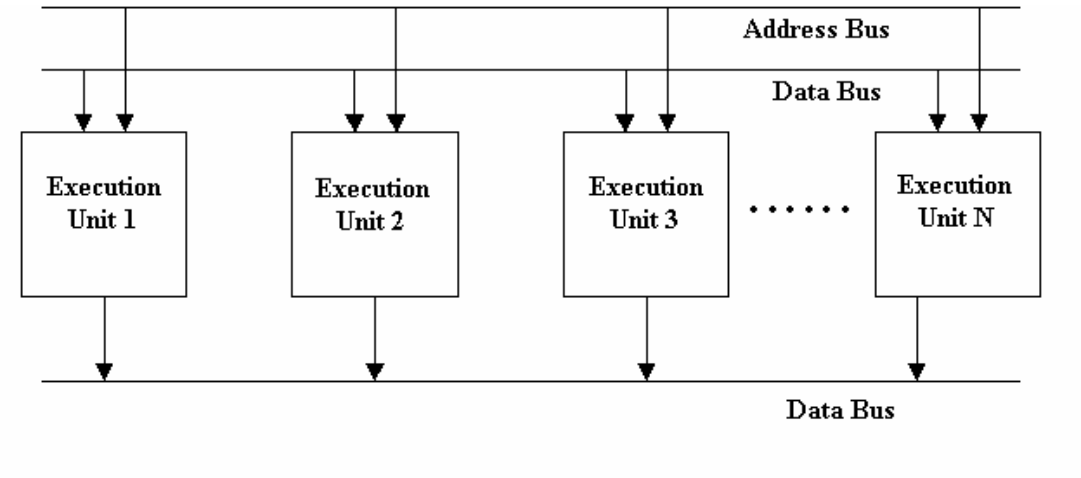


Fig 2.5

(A parallel unit)

However, parallel machines have a limitation, they cannot execute interlocked instructions (instructions with dependencies) in parallel [5, 6]. They are executed in serial like any serial machine. It takes more than one cycle to execute multiple instructions causing performance degradation in the machine. In addition, there is hardware underutilization as a result of serial execution in the parallel machine.

To improve performance it would be necessary to be able to execute these interlocked instructions in one cycle. Thus, interlock collapsing execution units in the form of multi-operand ALU's have to be employed.

2.1.5 OBJECTIVE ::

The thesis proposes design and simulation of a 32-bit 3-1 Interlock Collapsing ALU (ICALU), to allow the execution of two interlocked instructions in a single instruction cycle. This will improve the performance when it is degraded by data hazards. The device will be studied to find out if it meets it's objective which is to execute two interlocked instruction in one instruction cycle. The collapsing of interlocks will be confined to arithmetic and logical operations, on fixed point two's complement numbers.

2.1.6 INTERLOCKING IN PARALLEL COMPUTERS ::

WHAT ARE INTERLOCKED INSTRUCTIONS?

Instructions are said to be interlocked if an instruction depends on a previous instruction for its data so that they cannot be executed simultaneously. Consider the instruction pair of Fig. 2.6a.

i) ADD R2, R1 ; [R2] ← [R2] + [R1]
ii) ADD R3, R2 ; [R3] ← [R3] + [R2]

Fig 2.6a
(An interlocked instruction pair)

Instruction 2 required the result of instruction 1 (stored in R2). Instruction 2 can be executed only after instruction 1 has been executed. Thus, instruction 2 is said to be dependent on instruction 1. The dependency prevents the simultaneous execution of the instructions.

i) ADD R1, R2 ; [R1] ← [R1] + [R2]
ii) ADD R4, R3 ; [R4] ← [R4] + [R3]

Fig 2.6b
(A non-interlocked instruction pair)

Fig 2.6b is an example of a non-interlocked instruction pair. Instruction 2 does not require that instruction 1 be executed before it (instruction 2) is executed. Thus they can be executed simultaneously. Before moving on to the ICALU, consider

the data flow of a parallel machine, which can execute two instructions concurrently

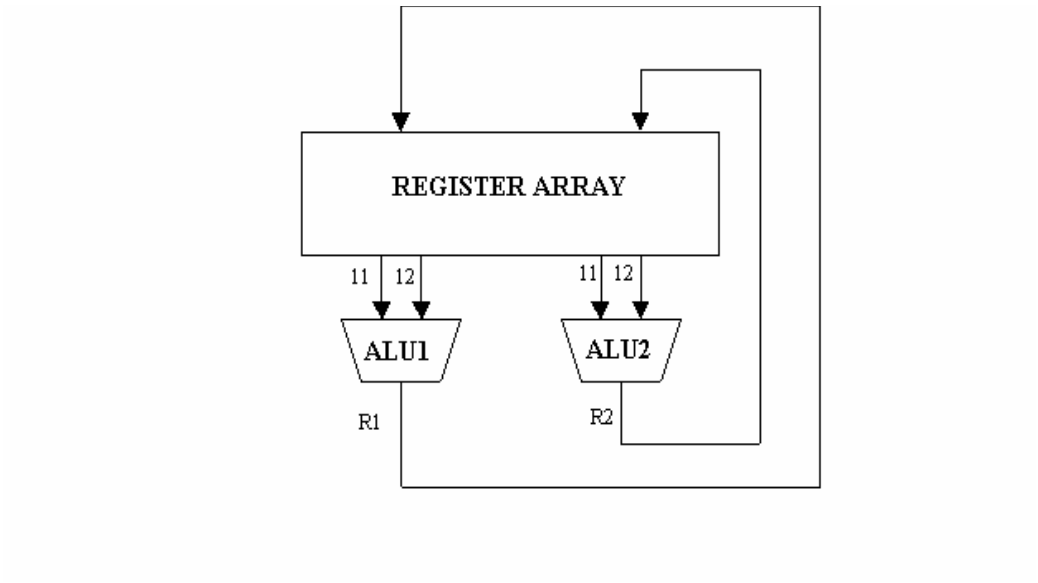


Fig 2.7

(Simplified view of the ALU unit for an ordinary parallel machine)

Fig 2.7 shows the ALU unit for a parallel machine which consists of two 2-1 ALU's. The notation '2-1' stands for two input operands and a single output (result). A 2-1 ALU has one 2-1 CLA (Carry Look Adder) to perform arithmetic operations and one logic stage to perform logical operations. The following explains the operation of the unit for both types of instructions.

I) NON-INTERLOCKED ::

ALU1 executes the first instruction and ALU2 executes the second simultaneously. Thus, the total execution time is one cycle.

II) INTERLOCKED ::

Since, an interlocked instruction cannot be executed simultaneously, ALU1 executes both the instructions one after the other requiring two cycles.

To resolve these interlocks a solution had been proposed previously. This can be shown in Fig 2.8 in which the proposed dataflow of a implementation for relieving fixed point data dependency interlocks is shown.

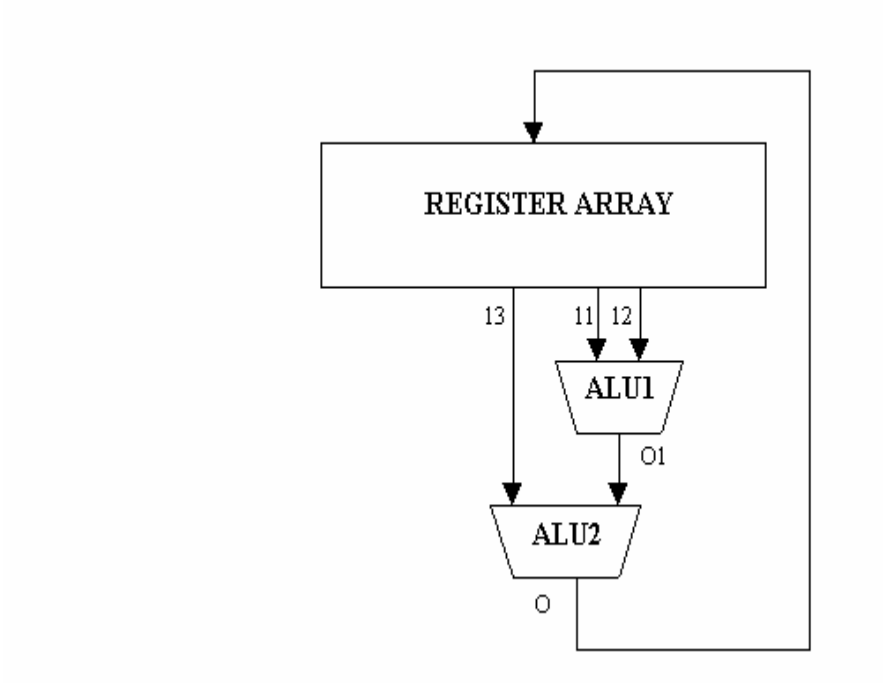


FIG 2.8

(WM'S APPROACH TO COLLAPSING INTERLOCKS)

Two ALU's are concatenated as shown. It can result in the execution of a multi-operation instruction, however, it requires twice the execution time of a single ALU operation. An attempt to execute the interlock in a cycle could result in an increase in the cycle time of the machine and unnecessarily penalize all instruction executions, resulting in practically no performance gain.

2.1.7 THE ICALU ::

In order for an implementation to eliminate interlocks between instructions and to execute such instructions in parallel (in addition to execution of non-interlocking instructions in parallel), it is required to collapse the interlocks with the incorporation of

- Multiple execution units, and
- Multi-operand execution units.

Multiple execution units are required because more than one instruction is being executed at a time. The number of instructions that can be executed is assumed to be two here and hence the number of execution units (ALU's) is two.

Multi-operand execution units are required, since there are two interlocked sequential instructions. The first instruction is execute by a traditional ALU. Since the second instruction may be dependent on the first, the second ALU must be capable of performing the collapsed instruction of both the instructions, in parallel to the first ALU. The second ALU has three input operands, one in addition to that of first ALU.

1) DESCRIPTION AND WORKING ::

The ICALU is basically a 3-1 ALU. It has 3-1 CSA (Carry Save Adder) in addition to the 2-1 CLA to achieve the desired 3-1 arithmetic operation. The ICALU also has an extra logic when compared to the 2-1 ALU. The ICALU is implemented in the parallel machine by replacing ALU2 with the ICALU. The operation of the parallel machine employing the ICALU is explained as ::

i) NON-INTERLOCKED ::

The operation is the same as that for the parallel machine when the sequence is non-interlocked.

ii) INTERLOCKED ::

Consider the interlocked sequence of Fig2.6a. ALU1 executes the first instruction as usual. The ICALU collapses the two instructions into a single 3-operand instruction as shown in Fig 2.9 :

ADD R3, R2, R1 ; [R3] ← [R3] + [R2] + [R1]

Fig 2.9

(The Collapsed Instruction)

Thus, the above instruction is executed in a single cycle by the ICALU. In short, the ICALU operates on both the instructions when there is interlock and on the second when there is no interlock.

The design of ICALU is described and simulated in VHDL. VHDL is a language to describe or model hardware systems. The next section of this chapter gives a brief explanation on the same.

iii) LIMITATIONS:

At percentage of interlocked instructions(X) ≈ 3%, the gain of the machine with ICALU is zero. Below this point the gain is negative, that is the machine with ICALU is slower than the machine Non-ICALU machine.

2.1.8 VHDL ::

VHDL is an acronym for VHSIC Hardware Description Language. The acronym VHSIC, in turn, stands for Very High Speed Integrated Circuit program. VHDL is a high level programming language, used for describing digital systems, just like any other conventional programming languages, such as C and Pascal, are used for computing mathematical functions or manipulating data. Execution of a VHDL program results in a simulation of the digital system.

WHY USE VHDL ?

With VHDL, we can quickly describe and synthesize circuits of five, ten, or twenty thousand gates. Equivalent designs with schematics or Boolean equations at the register transfer level can require several months of work by one person. In addition VHDL provides the capabilities described below ::

1) VHDL CAPABILITIES ::

i) POWER AND FLEXIBILITY ::

VHDL has powerful language constructs with which to write succinct code descriptions of complex control logic. It also has multiple levels of design descriptions for controlling design implementations. It supports design libraries and reusable components.

ii) TECHNOLOGY-INDEPENDENT DESIGN ::

VHDL permits us to create designs without having to first choose the technology. With one design description many technologies can be targeted.

iii) PORTABILITY ::

Because VHDL design description is a standard, your design descriptions can be taken from one simulator to another, one synthesis tool to another and one platform to another. As a result VHDL design descriptions can be used in multiple projects.

iv) MODELING STYLES ::

Supports both modeling styles, behavioral and structural

v) DESIGN METHODOLOGIES ::

It supports various design methodologies such as top-down, bottom-up and mixed designs.

2) LIMITATIONS ::

i) Too wordy.

ii) Debugging is difficult.

iii) Logic implementations created by synthesis tools may not always be efficient.

iv) Synthesis varies from tool to tool.

3) AN OVERVIEW OF VHDL MODELING STYLES ::

As mentioned earlier VHDL supports all the three modeling styles :

- Behavioral
- Structural
- Data flow

i) BEHAVIORAL :

A behavioral description explicitly defines the input/output function by specifying some sort of mathematical transfer function. For e.g., consider the Boolean equation which needs to be implemented.

F = A . B + C . D

A behavioral statement to implement the above equation would be :

F <= (A and B) or (C and D)

It is clear from the above statement that the structure of implementation is not known, until after synthesis is done. A behavioral description defines what the system does, but it does not necessarily indicate the design is to be implemented.

ii) STRUCTURAL :

In contrast, a structural representation or modeling style describes a digital system by specifying the interconnection of components that comprise the module. To implement the same equation as above, the structural description would be :

```
A1 : AND2_IP port map ( A, B, INT1 ) ;
A2 : AND2_IP port map ( A, B, INT2 ) ;
O1 : OR2_IP  port map ( INT1, INT2, Z ) ;
```

The statements clearly imply structure of the model which has an AND stage followed by an OR stage.

Chapter 3

PRELIMINARY DESIGN ISSUES OF ICALU

Functional requirements
Instruction category
Working of icalu

3.1) PRELIMINARY DESIGN ISSUES OF ICALU :

As discussed earlier, the ICALU performs a 3-1 operation in case of an interlocked sequence and a normal 2-1 operation in case of non-interlocked sequence. To design the ICALU we start with its functional requirements.

3.1.1) FUNCTIONAL REQUIREMENTS OF THE ICALU :

The functional requirements of the ICALU is divided into 2 modes :

1) Interlocked mode.
2) Non-interlocked mode.

1) MODE 1 (INTERLOCKED MODE) :

In Mode 1, the ICALU is in the Interlock mode, where it performs a three operand operation. Now, consider again an interlocked pair.

i) ADD A, B ; [A] ← [A] + [B]
ii) ADD A, C ; [A] ← [A] + [C]

Fig. 3.1a

(An interlocked instruction pair)

In Fig 2.10a an arithmetic operation follows an arithmetic operation. Similarly, there are other ways by which instruction can combine. The possible ways are categorized as follows:

i) Arithmetic followed by Arithmetic.
ii) Logical followed by Arithmetic.
iii) Arithmetic followed by Logical.
iv) Logical followed by Logical.

I) CATEGORY 1 : (ARITHMETIC FOLLOWED BY ARITHMETIC)

This category is represented by :

(A ± B ± C)

'±' addition or subtraction operation.

A – Operand 1

B – Operand 2

C – Operand 3

e.g., ADD A,B

SUB A,C

II) CATEGORY 2 : (LOGICAL FOLLOWED BY ARITHMETIC)

This category is represented by :

(A LOP B) ± C

('LOP' – Logical Operation)

e.g., AND A, B

ADD A, C

III) CATEGORY 3 : (ARITHMETIC FOLLOWED BY LOGICAL)

This category is represented by :

(A ± B) LOP C

e.g., ADD A, B

AND A, C

IV) CATEGORY 4 : (LOGICAL FOLLOWED BY LOGICAL)

This category is represented by :

(A LOP B) LOP C

e.g., AND A, B

XOR A, C

2) MODE 2 (NON-INTERLOCK MODE) :

In Mode 2 the ICALU is in the 'Non-Interlocked' mode. It performs a two operand operation. Consider a Non-Interlocked pair.

i) ADD A, B [A] ← [A] + [B]

ii) ADD D, C [D] ← [D] + [C]

Fig 3.1b

(An non-interlocked instruction pair)

Since no interlock exists between the two instructions, no collapsing is required. Thus ICALU executes only instruction 2. The categories in this mode are

i) Arithmetic.

ii) Logical

I) CATEGORY 1 (ARITHMETIC) :

This category is represented by :

A ± B

This category can be executed as a Mode 1 – Category 1 instruction if the third operand is forced to zero. It can be illustrated as :

(A ± B ± C) = (A ± B), when C = 0.

e.g., ADD A, B / Executed by ALU1/

SUB C, D

II) CATEGORY 2 (LOGICAL) :

This category is represented by :

A LOP B

Similarly this category can be executed as Mode 1 – Category 2 instruction, by forcing the third operand, C, to zero, as illustrated below :

(A LOP B) ± C = (A LOP B); when C = 0

e.g., ADD A, B / Executed by ALU1 /

AND C, D

From the above discussion it follow that :

- Category 1 is a 3-1 arithmetic operation, which requires a 3-1 adder. This can be achieved by cascading a 3-1 CSA followed by a 2-1 CLA.
- Category 2 & 4 require a logic stage before the adder stage. This is the Pre-CLA Logic Block.
- Categories 3 & 4 require a logic stage after adder stage. This is the Post-CLA Logic Block.
- The Mode 2, Non-interlocked operations is just a subset of Mode 1, Interlocked operations. They are executed as Mode 1 operations by setting the third operand to zero and hence do not require any additional circuitry within the ICALU.

Taking into consideration, all the above requirements, a logical dataflow model of the ICALU is developed. It is shown in Fig 2.11.

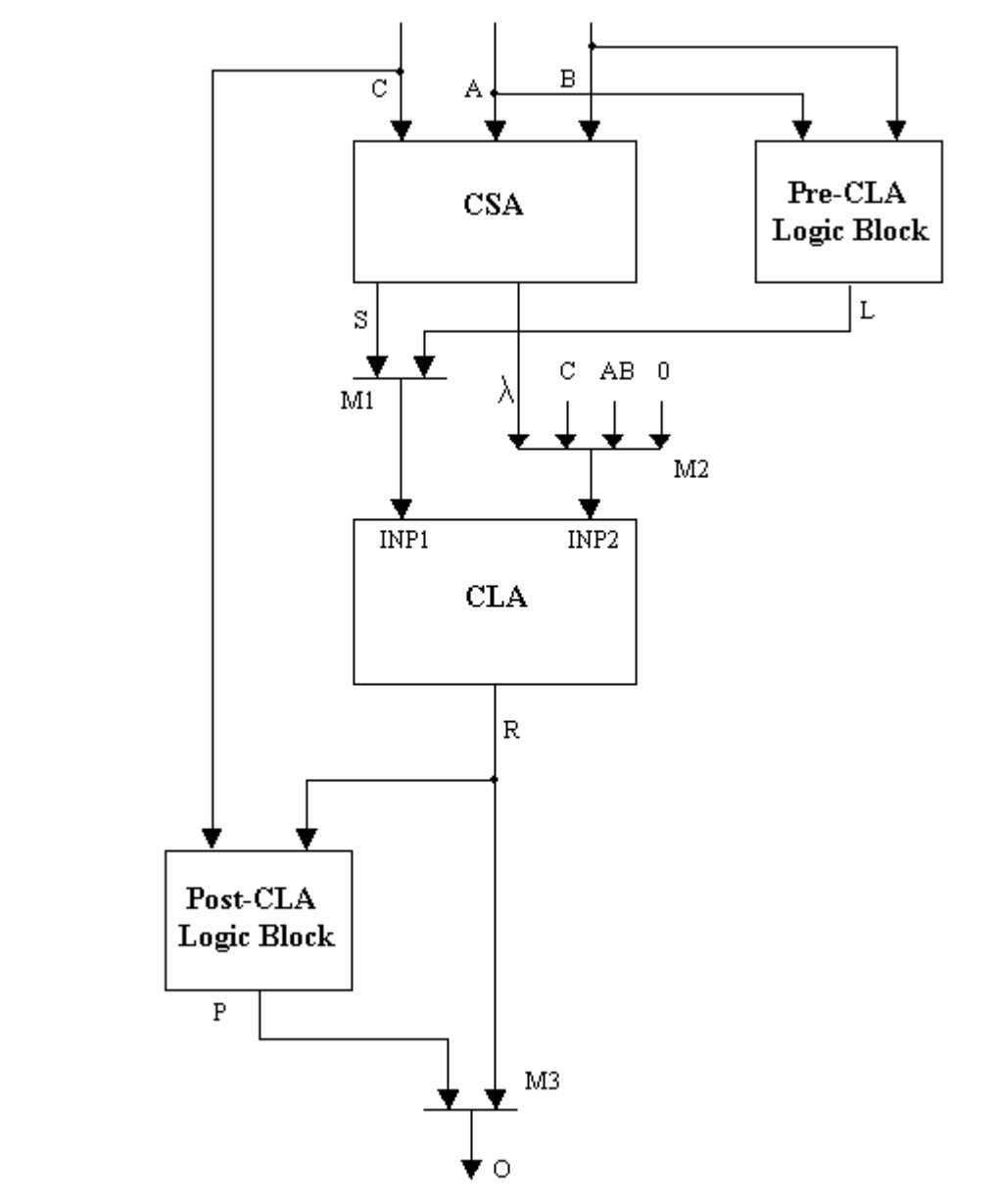


Fig 3.2

(Data flow form of ICALU)

3.3 WORKING OF THE ICALU MODEL :

3.3.1 MODE 1 :

I) CATEGORY 1 : (A ± B ± C)

The first category is executed by passing the operands through the CSA. The output S, which is the partial sum of the addition process, is selected by multiplexer M1 as the first input to the CLA. λ, which is the carry generated as a result of addition in the CSA, is selected by M2 to be the second input of CLA. The output of CLA, R, is the required sum. Since there is no post logical operation, the sum is bypassed to the output of the ALU, O. The 3-1 addition process is explained in detail in later chapters.

II) CATEGORY 2 : (A LOP B) ± C

The logical operation A LOP B is executed by the Pre-CLA Logic block. Signals L and operand C are the two inputs for CLA. R is bypassed to output as in Category 1.

III) CATEGORY 3 : (A ± B) LOP C

A ± B is 2-1 addition. The CSA is not required here. This operation is executed by selecting the output of Pre-CLA Logic Block L (A XOR B) as INPUT 1 and λ (A AND B) as INPTU 2 to the CLA. R is the required sum. Since there is a post logical operation, C LOP R, R cannot be bypassed to the output. Instead it is combined with C in the Post CLA Logic Block, to obtain the necessary logical operation. 'P' is now the output of ICALU.

IV) CATEGORY 4 : (A LOP B) LOP C

First and second inputs to CLA are L and zero respectively. The output of CLA is L again, since addition with zero gives the same number. The next step C LOP R is same as that in Category 3.

3.3.2 MODE 2 :

As explained earlier, Mode 2 operations are special cases of Mode 1. The Working for both categories remains the same for Mode 2, except that the third operand is set to zero. So, it is not explained any further.

3.4 DESIGN AND IMPLEMENTATION OF THE ICALU :

The ICALU is designed by first designing all the individual blocks. Implementation of each block is done by writing a VHDL program for that block. These individual blocks are finally assembled together, to work as one piece. This is also done by a VHDL program. An important thing to be noted is, the blocks in turn can have sub-blocks which are also assembled to form the blocks. Such a design methodology is called as 'hierarchical modeling'. This is similar to object oriented programming in high level languages such as ADA and C++. The next chapters will describe the design of each block and how to implement them using VHDL.

DESIGN OF VARIOUS MODULES

3.4.1 EXPRESSIONS FOR CLA INPUT STAGE

This chapter deals with the design of the Input Stage of CLA. Methods are discussed to obtain all the inputs for INPUT 2 of CLA from the carry output of CSA instead of obtaining them separately.

3.4.3DESIGN :

From the working of the ICALU model in previous chapter, the inputs to the CLA for both groups are summarized in tables below .

Category	**OPERATION**	***CLA Input 1***	**CLA INPUT 2**
1	C ± (A ± B)	A ⩛ B ⩛ C	(A .B ∧ B .C ∧ A .C)
2	C ± (A LOP B)	A LOP B	C
3	C LOP (A ± B)	A ⩛ B	A . B
4	C LOP (A LOP B)	A LOP B	0

TABLE 3.1A : INPUTS TO THE CLA FOR MODE 1 OPERATIONS.

‘⩛’ – represents logical XOR operation.

‘.’ - represents logical AND operation.

‘∧’ - represents logical OR operation.

CATEGORY	**OPERATION**	**CLA INPUT 1**	**CLA INPUT 2**
1	A ± B	A B	A . B
2	A LOP B	A LOP B	0

TABLE 3.1B : INPUTS TO THE CLA FOR MODE 2 OPERATIONS.

3.4.5 EXPRESSIONS FOR INPUT 1 :

From Table 3.1a, a 3-1 add operation is required for category 1. The CSA provides the input for this operation (S_i).

For categories 2, 3 and 4, the Pre-CLA Logic Block provides the input (L_i). Though category 3 is an arithmetic operation, Pre-CLA Logic Block provides the input, because sum of two operands, A and B, is given by A $\vee$ B which is a logical operation.

The selection between S_i and L_i can be achieved by a single control signal K_7 instead of a 2-1 Multiplexer M1. The equation for INPUT 1 is shown below :

$$\text{INPUT 1} = K_7 S_i + L_i \qquad (3.1)$$

'+' – represents Logical OR operation.

For category 1, $K_7 = 1$, thus,

INPUT 1 = 1, $S_i + 0 = S_i$,

For categories 2 to 4, $K_7 = 0$, hence,

INPUT 1 = 0, $S_i + L_i = L_i$

The control values for INPUT 1 are summarized in Table 3.2.

CATEGORY	**K_7**	**INPUT 1**
1	1	S_i
2	0	L_i
3	0	L_i

4	0	L_i

Table 3.2 : CONTROL VALUES FOR INPUT 1

3.4.6 EXPRESSIONS FOR INPUT 2 :

For INPUT 2 a method is discussed which eliminates the need for generating separate inputs, by obtaining all the inputs from the carry output of the CSA. Also the 4-1 Multiplexer for INPUT 2 can be eliminated by imbedding the selection in CSA.

The new expression for λ_{i+1} which is assigned to INPUT 2 can be written as :

$$\text{INPUT 2} = \lambda_{i+1} = K_2 A_i B_i + K_1 B_i C_i + K_1 A_i C_i + K_3 C_{i+1}. \quad \text{for } (1 \leq i \leq 31) \tag{3.2}$$

K_1, K_2 and K_3 are the three control signals given to the CSA to obtain the required combination of λ_{i+1} for the four different categories. These signals are produced in the decode cycle. The values of control signals to obtain the required input are shown below.

For category 1, $K_1 = K_2 = 1$, $K_3 = 0$, thus,
INPUT 2 = $\lambda_{i+1} = A_i B_i + B_i C_i + C_i A_i$.

For category 2, $K_1 = K_2 = 0$, $K_3 = 1$, thus,
INPUT 2 = $\lambda_{i+1} = C_i$

For category 3, $K_1 = 0$, $K_2 = 1$, $K_3 = 0$, thus,
INPUT 2 = $\lambda_{i+1} = A_i B_i$

For category 4, $K_1 = K_2 = K_3 = 0$, thus,

INPUT 2 = $\lambda_{i+1} = 0$

The control values for INPUT 2 are summarized in Table 3.3

K_1	**K_2**	**K_3**	**INPUT 2**
1	1	0	$(A_i B_i + B_i C_i + A_i C_i)$
0	0	1	C_i
0	1	0	$A_i B_i$
0	0	0	0

Table 3.3 : Control values for INPUT 2

3.5 PRE – CLA LOGIC BLOCK

This chapter deals with the design and reduction of the Pre-CLA Logic Block. The Pre-CLA Logic Block has to perform AND, OR, XOR & their inverts NAND, NOR, XNOR respectively. It also provides inputs to INPUT 1 of CLA for categories 2 to 4.

3.5.1 DESIGN :

To understand the logic stage lets assume the following control signals to the Pre-CLA Logic Block .

Control Signal	***Description***
K_{AND}	AND Inputs A & B
K_{OR}	OR Inputs A & B
K_{XOR}	XOR Inputs A & B
K_{INV}	Inverts above operations.

TABLE 3.4 : CONTROL SIGNALS TO PRE-CLA LOGIC BLOCK.

With these signals the expression for the output of the Pre-CLA Logic Block 'L' is expressed as :

$$L_i = A_i B_i K_{AND} K_{INV} + (A_i + B_i) K_{OR} K_{INV} + (A_i \veebar B_i) K_{XOR} K_{INV}$$
$$+ (A_i + B_i) K_{OR} K_{INV} + (A_i \veebar C_i) K_{XOR} K_{INV} + (A_i B_i) K_{AND} K_{INV}$$

where, $(0 \le i \le 31)$ (3.3)

It can further reduced to :

$$L_i = A_i B_i \overline{K_{AND} K_{INV}} + A_i \overline{K_{OR} K_{INV}} + B_i \overline{K_{OR} K_{INV}} + \overline{A_i B_i} \overline{K_{XOR} K_{INV}}$$

$$+ \overline{A_i B_i} K_{XOR} K_{INV} + A_i K_{AND} K_{INV} + B_i K_{AND2} K_{INV2}$$

$$+ \overline{A_i} \overline{B_i} K_{OR} K_{INV} + A_i B_i K_{XOR} K_{INV} + \overline{A_i} \overline{B_i} K_{XOR} K_{INV}$$

where, ($0 \leq i \leq 31$) (3.4)

The above expression is impractical to implement and it causes delay. However, if the operands are supplied to the ICALU as specified in the Table 3.5, the expression for L_i can be reduced to meet our requirement. Such a requirement does not add to the critical path because inversion of operands is required to execute subtraction and multiplexing operation for single operand functions.

FUNCTION	**ACTIVE CONTROLS**	A_I	B_I
AND	$\overline{K_{AND} K_{INV}}$	T	I
OR	$\overline{K_{OR} K_{INV}}$	T	T
XOR	$\overline{K_{OR} K_{INV}}$	T	T
AND-INV	$K_{AND} K_{INV}$	I	I
OR-INV	$K_{OR} K_{INV}$	T	I
XOR-INV	$K_{XOR} K_{INV}$	T	I

TABLE 3.5 : INPUT SPECIFICATION OF ALU FOR LOGICAL OPERATIONS.

These input specifications (3.4) can be expressed as :

$$L_i = L_{li} (K_{OR} \overline{K_{INV}} + K_{AND} \overline{K_{INV}}) + L_{ri} (K_{OR} \overline{K_{INV}} + K_{AND} \overline{K_{INV}})$$
$$+ L_{li} L_{ri} (K_{AND} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}})$$
$$+ L_{li} L_{ri} (K_{OR} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}}), \quad (3.5)$$

L_{li}, L_{ri} – The new left & right inputs respectively to the Pre-CLA Logic Block at bit position i

Further reduction is possible by defining three new controls that area combination of discrete controls as :

$$K_4 = K_{OR} \overline{K_{INV}} + K_{AND} \overline{K_{INV}} \quad (3.6)$$

$$K_5 = K_{AND} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}} \quad (3.7)$$

$$K_6 = K_{OR} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}} + K_{XOR} \overline{K_{INV}} \quad (3.8)$$

These controls can be directly produced from the decode of the instruction. Thus they can potentially be produced either in the decode cycle, in parallel with the register access, or they could be determined during the setup of the device (required for subtraction, sign extension, choosing between operations etc.). In either case these control signals do not contribute to the critical path of the ICALU. Substituting them into the expression for L_i produces:

$$L_i = L_{li}\,K4 + L_{ri}\,K4 + L_{li}\,L_{ri}\,K_5 + L_{li}\,L_{ri}\,K_6 \qquad (3.9)$$

Expressing the logical function block in this fashion reduces the delay. The input specifications that yield the required functions are tabulated in Table 3.6.

FUNCTION	ACTIVE CONTROLS	L_{li}	L_{ri}	K_{AND}	K_{OR}	K_{XOR}	K_{INV}	K_4	K_5	K_6	L_i
AND	$K_{AND}\,\overline{K_{INV}}$	A_i	$\overline{B_i}$	1	0	0	0	0	1	0	A_iB_i
OR	$K_{OR}\,\overline{K_{INV}}$	A_i	B_i	0	1	0	0	1	0	0	$A_i + B_i$
XOR	$K_{XOR}\,\overline{K_{INV}}$	A_i	B_i	0	0	1	0	0	1	1	$A_i\,\overline{B_i} + \overline{A_i}\,B_i$
AND-INV	$K_{AND}\,K_{INV}$	$\overline{A_i}$	$\overline{B_i}$	1	0	0	1	1	0	0	$\overline{A_i} + \overline{B_i}$
OR-INV	$K_{OR}\,K_{INV}$	A_i	$\overline{B_i}$	0	1	0	1	0	0	1	$\overline{A_i\,B_i}$
XOR-INV	$K_{XOR}\,K_{INV}$	A_i	$\overline{B_i}$	0	0	1	1	0	1	1	$\overline{A_i\,B_i + \overline{A_i}\,B_i}$

Table 3.6 : Output from reduced logic function block.

3.6 BINARY ADDERS AND ARITHMETIC

The arithmetic unit of the ICALU is implemented by binary adders. An introduction to binary adders is necessary. Binary subtraction and two's complement numbers are also explained in this chapter.

3.6.1 BINARY ADDERS:

1) FULL ADDER :

A full adder is the basic building block for all the adders. It adds three input bits. Two of the significant bits and the third bit is the carry bit from the previous stage. Thus it is called a full adder. Half adders are those that are those that add only 2 bits. The truth table and block diagram are presented below:

A	B	C_{IN}	S	C_{OUT}

0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.7 : Truth table of a Full Adder

The expressions for full adder are :

$$S = a \veebar b \veebar c \quad (3.10)$$

$$C_{out} = a\,b + b\,C_{in} + a\,C_{in} \quad (3.11)$$

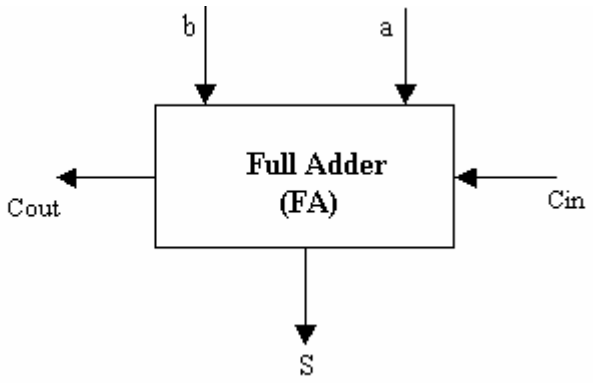


Fig 3.8 : A Full Adder

2) RIPPLE CARRY ADDER:

One of the most basic adders is the ripple carry adders. The addition is similar to that of paper and pencil addition. A block diagram to add two 4-bit binary numbers is shown in Fig2.2. The carry is allowed to ripple from one stage to another. However the

ripple carry is the slowest, because the carry has to propagate from the least significant bit(LSB) to the most significant bit(MSB). Hence it is not used for larger adders.

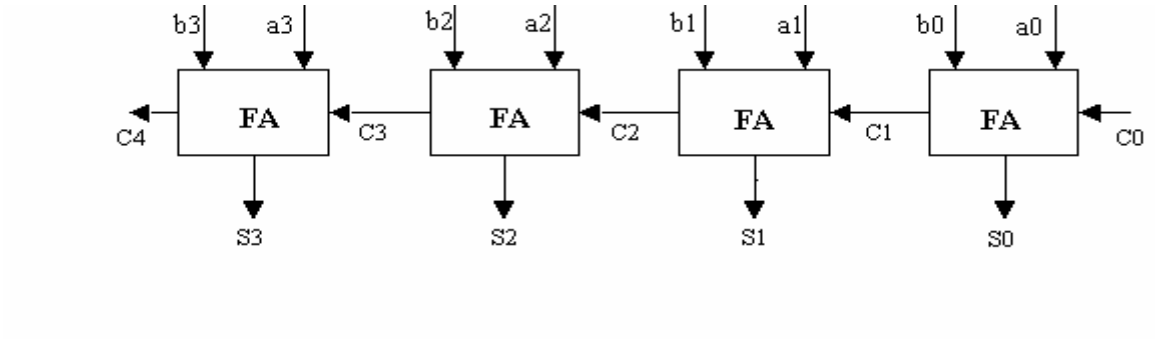


Fig 3.9: A 4-Bit Ripple Carry Adder.

3)CARRY LOOK AHEAD ADDER (CLA):

The CLA is faster than the ripple carry adder. The carry input to each stage is generated directly, instead of allowing the carry to ripple from one stage to another. Fig 5.3 shows the block diagram of a CLA. The Boolean expression for each carry block can be defined by using the carryout expression of a full adder. It is given as:

$$C_{i+1} = x_i\, y_i + C_i\, (\, x_i + y_i\,) \qquad (5.2)$$

Where, i=0,…….,N, and, N= number of bits in each number.

For example,the output of first carry block:

$$C_1 = x_0\, y_0 + C_0\, (\, x_0 + y_0\,)$$

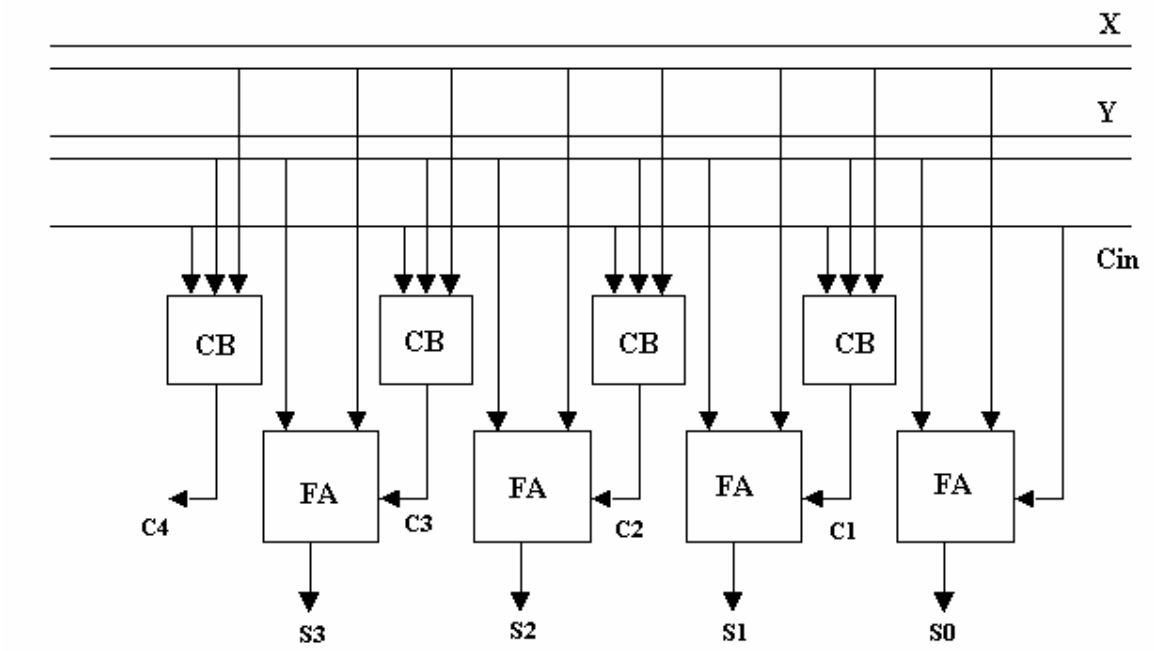


FIG 3.10 : A 4-BIT CARRY LOOK AHEAD ADDER.

4)CARRY SAVE ADDER (CSA):

The CSA is used when more than two numbers are to be added.

For example, Consider addition on three numbers (X,Y,Z).

	0101	X
	0011	Y
+	0100	Z
	————	
	0010	Partial sum
	1010	Saved Carry

In the next step, the sum and saved carry are added with each other.

	0010	Partial Sum
+	1010	Saved carry
	————	

1010 Final sum

In the last step the CLA is used to add the partial sum with saved carry.

Fig 3.4 shows the block diagram of the addition process. The first stage is the CSA. The carry consists of a chain of full adders. A full adder is present for each significant bit position. Unlike the ripple adders, in carry look ahead adders carry is saved for the next stage.

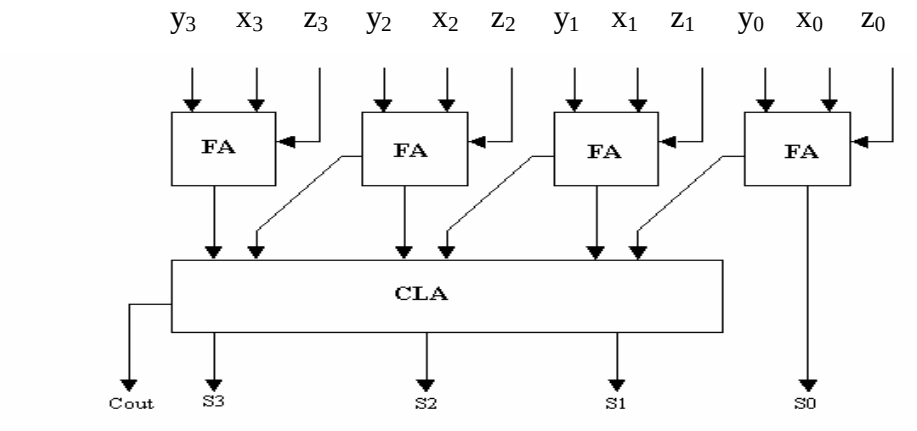


FIG 3.11 : A 4-BIT CARRY SAVE ADDER

3.6.2 BINARY SUBTRACTION AND TWO'S COMPLEMENT:

Binary subtraction is achieved by representing negative binary numbers in some form. There are many schemes for representing negative binary numbers like the sign magnitude, one's complement, two's complement etc. The most popular scheme is the two's complement representation as it is the most convenient method.

1) TWO'S COMPLEMENT REPRESENTATION :

A binary number can be represented in two's complement as:

$$\text{Two's complement} = \begin{cases} 2^N - B, B \neq 0 \\ 0, \quad B = 0 \end{cases} \qquad (3.12)$$

The above equation is rearranged as :

Two's complement = $(2^N - 1) - B + 1, \quad B \neq 0$

$0, \quad B = 0$ (3.12a)

Noting that two's complement of an N-bit binary number B can be found by subtracting each bit of a binary integer B from 1 and then adding 1 to the total N-bit resulting difference. Moreover, since 1 – 0 = 1 and 1 – 1 = 0, subtracting each bit form 1 is equivalent to simply flipping or inverting each bit. For example two's complement of –5 can be obtained as :

5 =	0101	
	↓	
	1010	Inverted bits
+	1	Add 1
	————	
-5 =	1011	Two's complement of –5

Two's complement numbers can be easily generated by cascading an inverter stage with an adder stage.

The decimal range for an N-bit binary number is given as :

$(-2^{N-1} \text{ to } 2^{N-1} - 1)$

For N=32, the decimal range is given by :

$(-2^{31} \text{ to } 2^{31} - 1) = (-2147483648 \text{ to } 2147483747).$

2) TWO'S COMPLEMENT SUBTRACTION AND ADDITION :

Two's complement addition and subtraction are very similar to standard binary addition. It is illustrated by the following examples :

A) ADDITION :

	4		0100
+	5	⟶	0011
	———		———

7 0111

0111 is the correct two's complement representation of 7.

B) SUBTRACTION :

3 0011

\- 7 ⟶ 1011

——— ———

\- 4 1100

1100 is the correct two's complement representation of –4.

However, there are two situations the two's complement addition differs from standard binary addition.

I) CARRY OUT FROM MSB :

In this exception, any carry outs from MSB are ignored.

EXAMPLE :

(-3) 1101

\+ (-4) 1100

——— ———

(-7) ~~1~~1001

1001 is the correct two's complement representation for –7. Note that the carry has been eliminated.

II) OVERFLOW / UNDERFLOW :

Overflow is said to occur when an arithmetic operation yields a result that is greater than the range's positive limit of ($2^{N-1} - 1$).

3.7 DESIGN OF CSA STAGE

In the previous chapters various adders were discussed. The CSA was also explained. The CSA stage for the ICALU not only has to generate the sum and carry for the next stage, but also the inputs for INPUT 2 of CLA. This is designed and implemented in this chapter.

3.7.1 DESIGN:

In the last chapter we saw the block diagram for three 4-bit inputs using a CSA . It can be extended to 32 bits. The equation for sum and carry are:

$$SUM = S_i = A_i \veebar B_i \veebar C_i \qquad \text{for } (0 \leq i \leq 31) \qquad (3.13a)$$

$$CARRY = \lambda_{i+1} = A_i B_{i+} B_i C_{i+} Ai C_{i,} \qquad \text{for } (1 \leq i \leq 31) \qquad (3.13b)$$

$A_{i,} B_{i,} C_i$ are the i^{th} bits of operands A, B & C respectively.

As explained in chapter 3 the carry out of CSA is designed to provide all inputs for INPUT 2 of CLA. It is given as:

$$\lambda_{i+1} = K_2 A_i B_i + K_1 B_i C_i + K_1 A_i C_i + K_3 C_{i+1,} \qquad \text{for}(1 <= i <= 31) \qquad (3.14)$$

3.7.2 INTEGER RANGE OF ARITHMETIC OPERATION:

The maximum range that can be handled by the 32-bit arithmetic unit, with two's complement representation:

$= (-2^{31}$ to $2^{31}-1) = (-2147483648$ to $2147483647)$.

3.7.3 IMPLEMENTATION:

The (3.13a) and (3.14) are bit-wise expressions. They are the basic building blocks of the CSA. First, they are implemented as individual components (blocks). Later, they are instantiated the required number times to obtain the CSA.

1) IMPLEMENTATION OF SUM (S_I):

The VHDL code for SUM3_1 is given in Appendix A.6.2. It creates the component SUM_1.

2) IMPLEMENTATION OF CARRY(Λ_{I+1}):

The VHDL code for CSA_CARRY is given in Appendix A.6.3. It creates the component CSA_CARY.

3) IMPLEMENTATION OF CSA :

The entities SUM3_1 and CSA-CARY are the basic building blocks of the CSA. The sum is generated for bit positions 0 to 31 where as the carry is from 0 to 30. Since, the additions considered are in 2's complement the final carry (i.e., bit-position 31) is discarded. Thus, carry spans one bit positions lesser when compared to sum. The

component SUM3_1 is instantiated 16 times for each bit position to obtain sum. The 'for generate' statement is used to repeat the instantiations for the desired number of times. The program is shown in the Appendix A.6.1.

3.8) DESIGN OFCLA STAGE

The basic of a CLA was explained in chapter 5. using the same principle, it is extended to 32 bits in this chapter.

3.8.1)Carry expressions for CLA:

Consider again, equation 3.11, which is the Boolean expressions for a CLA carry block:

$$C_{i+1} = x_i\, y_i + C_i\, (\, x_i + y_i\,) \qquad (3.15)$$

To simplify equation 3.15, notation g and p are defined as;

$$g_i = x_i\, y_i,$$
$$p_i = x_i + y_i$$

On substitution, expression 3.15 reduces to:

$$C_{i+1} = g_i + C_i\, p_i \qquad (3.16)$$

The notation g stands for generating a carry.
Since output carry ($C_{i+1)}$ is 1 whenever g_i is 1.
The notation p stands for propagating input carry to the output carry.
Since C_{i+1} is 1 whenever P_i is 1.

Using these notations, we get

$C_1 = g_o + p_0 C_0$

$C_2 = g_1 + p_1 g_0 + p_1 p_0 C_0$

$C_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0$

$C_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0$

Consider the last carry block C_4. The number of OR terms are five and the last product term in C_4 also has five inputs.

In general, for n inputs:

The number of OR terms are n+1. Thus, to implement this we need an OR gate that has a fan in of at least n+1. Similarly, the fan in of AND gate should be at least n+1. Thus if the fan in of a particular technology is n, then it may not be possible to implement a given block directly.

In addition, we can clearly see each carry block expression is different from the other. Thus, a modular design is not possible for a large n. A modular design requires a structure in which similar parts can be used.

Thus to solve the preceding problems, we limit fan-in and fan-out to a given number depending on the technology. The result is an adder with a large n, broken into many smaller adders cascaded together. For example, an 8-bit CLA with fan in limited to say, four, can be implemented by cascading two 4-bit CLA's (Fig 3.3) together.

This can be done by defining two new terms, denoted as, G_0 – Group propagate and P_0 – group generate, where,

$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

$P_0 = p_3 p_2 p_1 p_0 C_0$

Thus we can express C_4 as :

$$C_4 = G_0 + P_0 C_0$$

In this group C_8, is computed similarly to C_4 :

$$C_8 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

where,

$$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 \text{ and,}$$

$$P_1 = p_7 p_6 p_5 p_4$$

1) The 32 – Bit CLA :

The maximum fan-in of the gates in the ICALU design is assumed to be 8. Assuming a modular design, a 32-bit CLA can be implemented as four 8-bit CLA's.

The additional carries for an 8-bit CLA, using the above notations are :

$$C_5 = g_4 + p_4 g_3 + p_4 p_3 g_2 + p_4 p_3 p_2 g_1 + p_4 p_3 p_2 p_1 g_0 + p_4 p_3 p_2 p_1 p_0 C_0$$

$$C_6 = g_4 + p_5 g_4 + p_5 p_4 g_3 + p_5 p_4 p_3 g_2 + p_5 p_4 p_3 p_2 g_1 + p_5 p_4 p_3 p_2 p_1 g_0$$
$$+ p_5 p_4 p_3 p_2 p_1 p_0 C_0$$

$$C_7 = g_5 + p_6 g_5 + p_6 p_5 g_4 + p_6 p_5 p_4 g_3 + p_6 p_5 p_4 p_3 g_2 + p_6 p_5 p_4 p_3 p_2 g_1$$
$$+ p_6 p_5 p_4 p_3 p_2 p_1 p_0 C_0$$

$$C_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 g_3$$
$$+ p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0$$
$$+ p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 C_0$$

Now, C8 can be represented in terms of group generate and propagate terms, G and P as:

$G_0 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7\ p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2$

$+ p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0$

$P_0 = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0$

G_0, P_0 are the outputs of CLA. They are externally combined with C_0 to obtain C_8. So, C_8 can be expressed as,

$C_8 = G_0 + P_0 C_0$.

Similarly,

$C_{16} = G_1 + P_1 C_8 = G_1 + P_1 G_0 + P_1 P_0 C_0$ and,

$C_{24} = G_2 + P_2 C_{16} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$

(G_0,P_0), (G_1,P_1), (G_2,P_2) are the outputs of CLA1, CLA2 and CLA3, respectively.
G_3 and P_3 are the discarded because the final carry out, that is C_{32} in two's complement is discarded.

3.8.2) Implementation:

The 32-bit CLA can be implemented by creating an 8-bit CLA first. Later, the 8-bit CLA' are connected together to obtain the 32-bit CLA.

1) Implementation of 8-bit CLA:

The 8-bit CLA is obtained using the preceding expressions. The sum block for the CLA are obtained by instantiating the component SUM3_1. It has already been discussed in chapter 6. There are eight sum and carry blocks for each of the 8-bit positions. Two CLA's have been implemented here, CLA_1 and, CLA_2.Though, both are 8-bit, CLA_1 is slightly different from the other CLA's, since it does not have an input carry. It has

been created as a separate component. The program for the 8-bit CLA is shown in Appendix A.5.

2 IMPLEMENTATION OF 32-BIT CLA:

The 32-bit CLA is obtained by installing the four 8-bit CLA's and generating the intermediate carries C_{16} and $C_{24,}$ appropriately. The final carry C_{32} is discarded, since all operations are in two's complement. The program for 32-bit CLA is shown in A.4.

3.9 DESIGN OF POST – CLA LOGIC BLOCK

This chapter deals with the design of the Post-CLA Logic Block of the ICALU. The Post-CLA Block performs logic operations between the third operand and the result of the operation on the first two operands. The Post-CLA Logic Block is not similar to the Pre-Block because the inverting inputs are not readily available.

3.9.1 DESIGN :

The control signal format remains the same as that for the Pre-CLA Logic Block. But the control unit generates a separate set of signals for the Post-CLA Logic Block.

Control Signal	***Description***
F_{AND2}	AND Inputs R_i & C_i
F_{OR2}	OR Inputs R_i & C_i

F_{XOR2}	XOR Inputs R_i & C_i
F_{INV2}	Inverts above operations.

Table 3.12 : Control signals to Post-CLA Logic Block.

R_i – Result from the CLA stage.

C_i – The third operand to ICALU.

The expression for the Post-CLA Logic Block is similar to (3.4), it is :

$$L_i = R_i C_i K_{AND2} \overline{K_{INV}} + R_i K_{OR} \overline{K_{INV2}} + C_i K_{OR2} \overline{K_{INV2}} + R_i C_i K_{XOR2} \overline{K_{INV2}}$$
$$+ \overline{R_i} C_i K_{XOR2} \overline{K_{INV2}} + R_i K_{AND2} \overline{K_{INV2}} + \overline{C_i} K_{AND2} K_{INV2}$$
$$+ \overline{R_i} \overline{C_i} K_{OR2} K_{INV2} + R_i C_i K_{XOR2} K_{INV2} + \overline{R_i} C_i K_{XOR2} K_{INV2}$$

where, ($0 \leq i \leq 31$) (3.17)

'+' – Logical OR operatin.

To eliminate the multiplexer M3, in Fig 2.2, we can add a new control signal F_{ADD} to the above expression as follows :

$$L_i = R_i C_i K_{AND2} \overline{K_{INV}} + R_i K_{OR} \overline{K_{INV2}} + C_i K_{OR2} \overline{K_{INV2}} + R_i C_i K_{XOR2} \overline{K_{INV2}}$$
$$+ \overline{R_i} C_i K_{XOR2} \overline{K_{INV2}} + R_i K_{AND2} \overline{K_{INV2}} + \overline{C_i} K_{AND2} K_{INV2}$$

$$+ R_i C_i K_{OR2} K_{INV2} + R_i C_i K_{XOR2} K_{INV2} + R_i C_i K_{XOR2} K_{INV2} + R_i F_{ADD2.} \quad (3.17a)$$

Now, L_i represents the output of the ICALU. The Table 3.9 summarizes the output of the Post-CLA Logic Block for different control values.

F_{ADD}	F_{AND}	F_{OR}	F_{XOR}	F_{INV}	OUTPUT (L_I)
1	0	0	0	0	R_i
0	1	0	0	0	$R_i C_i$
0	0	1	0	0	$R_i + C_i$
0	0	0	1	0	$R_i \veebar C_i$
0	1	0	0	1	$\overline{R_i C_i}$
0	0	1	0	1	$\overline{R_i + C_i}$
0	0	0	1	1	$\overline{R_i \veebar Ci}$

TABLE 3.13: OUTPUT TABLE OF POST-CLA LOGIC BLOCK.

Now, the control signals in (3.17a) are grouped to enable pre-calculation:

$$L_i = R_i \left(C_i K_{AND2} K_{INV2} + K_{OR} K_{INV2} + C_i K_{XOR2} K_{INV2} + F_{ADD2} \right)$$

$$+ R_i \left(K_{AND2} K_{INV2} + C_i K_{OR2} K_{INV2} + C_i K_{XOR2} K_{INV2} + C_i K_{XOR2} K_{INV2} \right)$$

$$+ \left(C_i K_{OR2} K_{INV2} + C_i K_{AND2} K_{INV2} \right). \qquad (3.18)$$

The delay of the Post-CLA Logic Block can be reduced by pre-calculating the following signals, since they rely only on the ICALU inputs :

$$PCLA1_i = C_i K_{AND2} K_{INV2} + K_{OR} K_{INV2} + C_i K_{XOR2} K_{INV2} + F_{ADD2} \qquad (3.19a)$$

$$PCLA2_i = K_{AND2} K_{INV2} + C_i K_{OR2} K_{INV2} + C_i K_{XOR2} K_{INV2} + C_i K_{XOR2} K_{INV2} \qquad (3.19b)$$

$$PCLA3_i = C_i K_{OR2} K_{INV2} + C_i K_{AND2} K_{INV2} \qquad (3.19c)$$

Substituting the above equations in (3.18), we get

$$L_i = R_i PCLA1_i + R_i PCLA2_i + PCLA3_i \qquad (3.20)$$

(3.20) is the output of the Post-CLA Logic Block. It is also represents the final output of the ICALU.

3.9.2 IMPLEMENTATION :

The implementation of Post-CLA Logic Block is also similar to earlier implementations, that is, CSA, Pre-CLA Logic Block, etc. First, the bit-wise component is implemented and later instantiated to obtain the Logic Block.

1) IMPLEMENTATION OF BIT-WISE LOGIC COMPONENT :

The bit-wise logic component, P_CLA_BCMP, is the implementation of (3.20). The program is shown in A.8.2.

2) IMPLEMENTATION OF POST-CLA LOGIC BLOCK :

The logic block is implemented by instantiating P_CLA_BCMP for bit positions 0 to 31. The program is shown in A.8.1. The program creates entity P_CLA_LOGBLK.

Chapter 4

ICALU DATAFW
MODELLO

THE INTERLOCK COLLAPSING ALU UNIT:

In this chapter all the designed components are put together to implement the ICALU. Also, ALU1 is created using the designed components. Finally, the Interlock collapsing ALU unit is implemented which consists of both ALU1 and ICALU. The chapter also estimates the relative delay.

4.1 REDUCED ICALU MODEL :

Resulting from the design of the various stages in the preceding chapters a reduced ICALU is obtained. The result was the elimination of the multiplexers M2 and M3 and also better implementations of the Pre and Post-CLA Logic Blocks. The block diagram is shown in Fig 4.1. The program for ICALU is in the Appendix A.2.

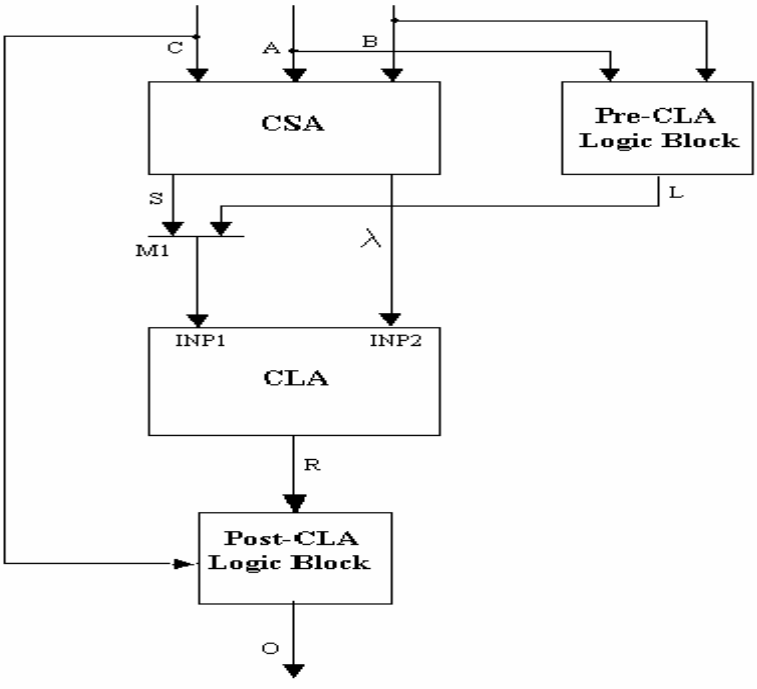


FIG. 4.2 : REDUCED DATAFLOW MODEL OF ICALU

4.2 ALU1 MODEL :

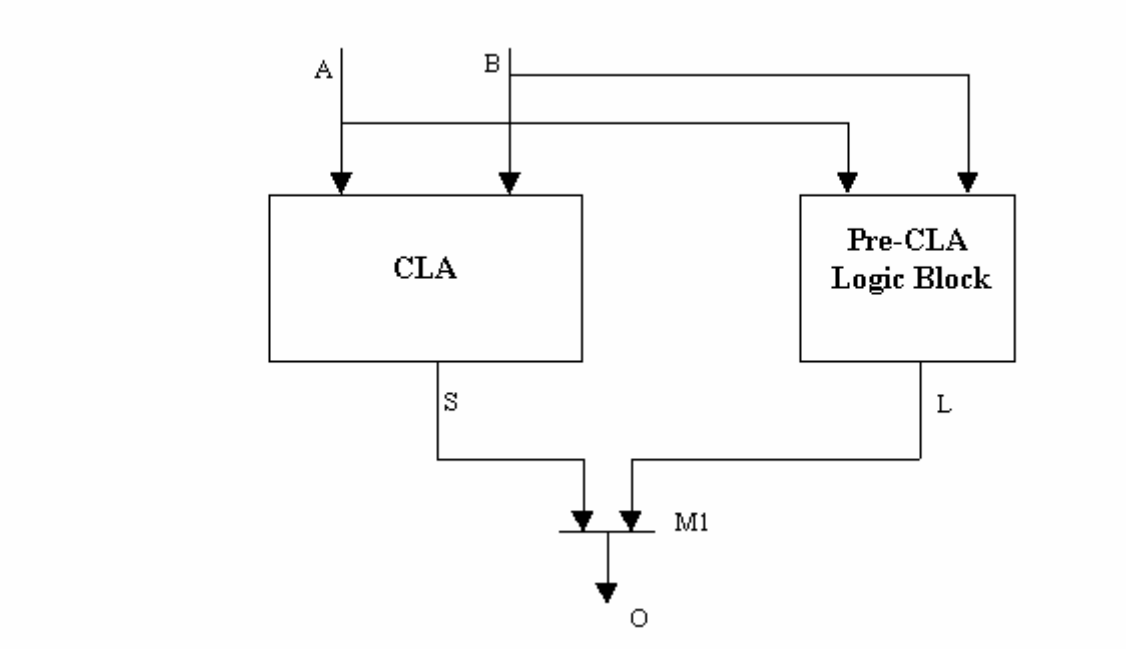


Fig 4.3

(Dataflow Model of ALU1)

The control signals for multiplexer are K_{12} and K_{13} and are set as follows :

I) CATEGORY 1 (ARITHMETIC) :

$K_{12} = 1$ and, $K_{13} = 0$;

Output of ALU1 = O = A ± B.

II) CATEGORY 2 (LOGICAL) :

$K_{12} = 0$ and, $K_{13} = 1$;

Output of ALU1 = O = A LOP B.

The values of control signals are summarized in Table 4.1 :

CATEGORY	K_{12}	K_{13}	O
1	1	0	A ± B
2	0	1	A LOP B

Table 4.1 : Output table for ALU1

4.2.2 IMPLEMENTATION :

The ALU1 is implemented using the block diagram above. The components CLA and PREBLK are the adder and the logic block respectively, for ALU1. The program for entity ALU1 is shown in A.3.

4.3 INTERLOCK COLLAPSING ALU UNIT :

The Interlock collapsing ALU unit consists of ALU1 and the ICALU operating in parallel. The block diagram of the Interlock collapsing unit was shown in Chapter 1, Fig 1.7. The program for entity ICUNIT is shown in A.2.

4.4 ESTIMATION OF RELATIVE DELAY BETWEEN ALU1 AND ICALU :

In this section the relative delay between the ALU1 in Fig 4.2 and the ICALU in Fig 4.1 is estimated. The relative delay is the difference between the delay of ALU1 and the ICALU. The delay is required to find out the instruction cycle length. The delay of a device can be estimated by taking a logic gate count from the input to the output. Only the delay between both ALU's considered because all other stages in their respective paths are identical, hence they also have identical delays.

Now, compare Fig 4.1 (ICALU) and Fig 4.2 (ALU1).
By elimination, it is deduced that the ICALU has two additional stages when compared to the ALU1 which are :

i) The CSA and,

ii) The Post-CLA Logic Block.

The procedure is :

1) The CLA and multiplexers are common to both the ALU's. Hence they can be eliminated.
2) The extra stages in the ICALU path are the CSA and the Post-CLA Logic Stage.
3) The Pre-CLA Logic stages are not considered because in case of ALU1 it is parallel with the CLA stage and has lesser stages than the same. Where as, in case of the ICALU it is in parallel and has the same delay as the CSA.

The logic delay of both stages are :

I) CSA :

To estimate this consider (3.13a) and (3.14) which represent the input-output transformations of the CSA sum and carry respectively. Both are in parallel.

$$SUM = S_i = A_i \text{ V } B_i \text{ V } C_{i,} \qquad (3.13a)$$

$$\lambda_{i+1} = K_2 A_i B_i + K_1 B_i C_i + K_1 A_i C_i + K_3 C_{i+1}. \qquad (3.14)$$

(3.13a) and (3.14) can each be implemented in one gate delay using custom-built CMOS libraries. A ± 3 X 4 AO gate can serve this purpose ('+' represents AND-OR and '-' represents AND-OR-INVERT). The delay of this gate is assumed to be 1 gate stage as that of any other gate in the assumed libraries.

II) LOGIC DELAY OF POST-CLA LOGIC BLOCK :

Similarly, (3.9) (shown below) can be implemented in one gate delay by the AO gate.

$$L_i = L_{li} K_{PRE1} + L_{ri} K_{PRE1} + L_{li} L_{ri} K_{PRE2} + L_{li} L_{ri} K_{PRE3} \quad (3.9)$$

Thus the total relative gate delay of the ICALU over the ALU1 =

Logic delay due to CSA stage + Logic delay due to Post-CLA Logic Stage =1+1 = 2.

4.5 DETERMINATION OF INSTRUCTION CYCLE LENGTHS OF A MACHINE WITH AND WITHOUT ICALU :

The average instruction length is calculated to find out the speed of the machine. The instruction cycle length varies for each instruction. Hence an average instruction length has to be calculated. It is sufficient to take the average of only frequently executed instructions. The following discussion shows how the instruction lengths can be calculated for a given instruction. But first, Fig 2.4 is redrawn again.

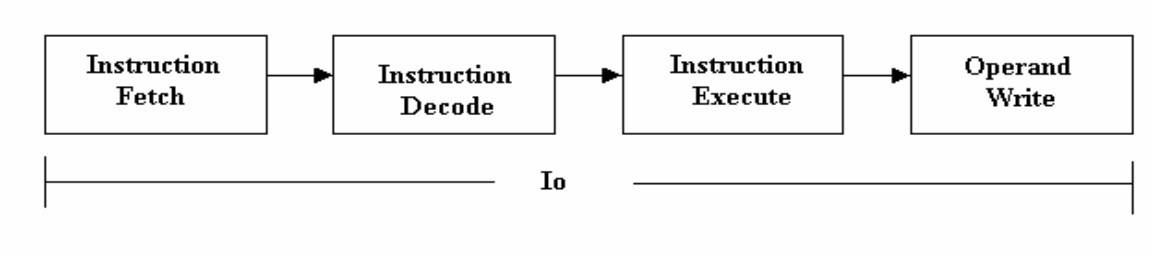


Fig 4.5

(Phases of Instruction execution process)

Fig 4.3 represents the instruction path of serial machine. The time to execute an instructions given as I_0, or the basic instruction cycle time. The individual stage have been discussed in Chapter 1.

4.4.1 Without ICALU :

For a parallel machine there are two such paths in parallel. Fig 4.4 shows instruction execution (considering non-interlocked case) in a parallel machine with respect to time.

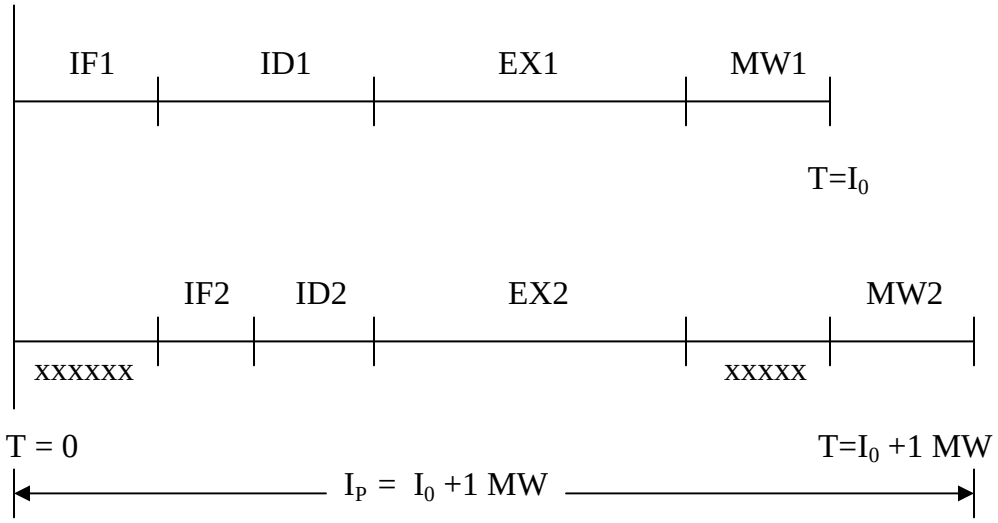


Fig 4.6

(Instruction cycle of a parallel machine without ICALU)

Fig 4.4 shows the instruction cycle of a parallel machine for a two-operand instruction pair shown below. The upper cycle in the figure represents execution of instruction 1. The instruction time is the same as the basic instruction cycle time, I_0. Execution of Instruction 2 is shown in the lower half. It starts a memory write cycle after the first instruction, because memory cannot be accessed simultaneously. It shifts to the right by 1MW. The x's in figure represents an idle cycle.

```
ADD   R1, R2        / Executed by ALU1 /
ADD   R3, R4        / Executed by ALU2 /
```

The ID2 is smaller than ID1 by one memory access because we already have R2, fetched by Instruction 1. This compensates for the delay in start of execution of Instruction 2 and thus the execution cycles of both the instructions start at the same time. After the EX cycle is complete, Instruction 2 has to wait for 1MW for Instruction 1 to complete its memory access.

Instruction 2 takes a further 1MW to complete its cycle. Thus from the figure it can clearly be seen that the instruction time of a parallel machine is lengthened by 1MW.

4.4.1 With ICALU :

The instruction cycle for a machine with ICALU is shown in Fig 4.5.

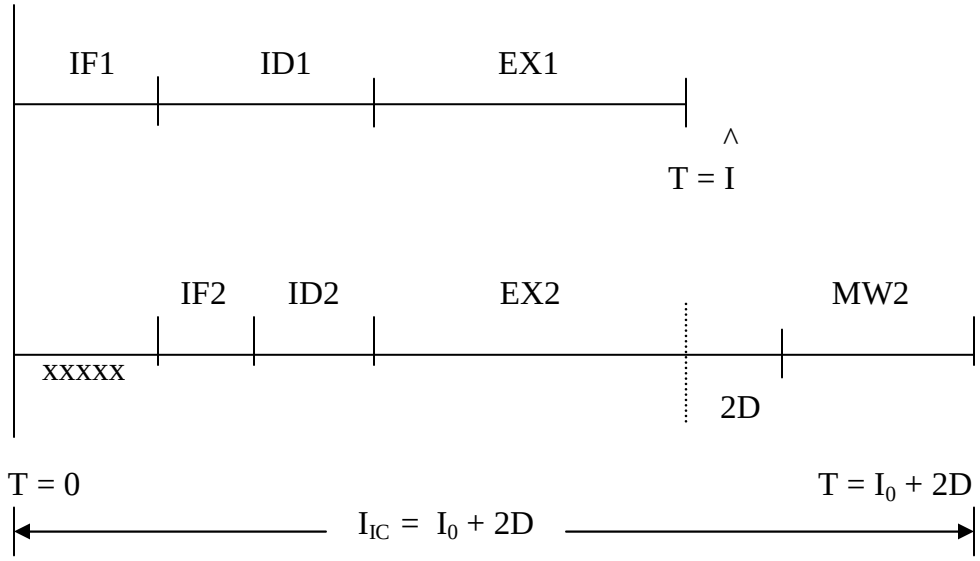


Fig 4.7

(Instruction cycle of Parallel Machine with ICALU)

The instruction cycle in figure is for the pair given below :

ADD R1, R2

ADD R1, R3

The operation is almost similar to that of an ordinary parallel machine except that there is no memory access for ALU1. Hence the memory access starts once the ICALU completes it's execution which is two additional logic or gate delays more than the 2-1 ALU. Hence its instruction cycle time increases to $I_0 + 2\ D$ (D – Unit gate delay or the delay of one gate).

MW can be treated as three gate delays for CMOS memories. Substituting this value average instruction length can be calculated.

Chapter 5

PERFORMANCE ANALYSIS

PERFORMANCE ANLAYSIS

In this chapter the performance of a Non-ICALU and that of a parallel machine with the ICALU is compared. Table 5.1 shows the average instruction lengths of a machine with ICALU and a Non-ICALU parallel machine for the interlocked and Non-interlocked categories. The average instruction lengths were calculated by taking the average of instructions lengths obtained for all possible interlocked and non-interlocked pairs (See Appendix B). The average instruction length is the time taken to execute an instruction pair, that is two consecutive instructions.

CATEGORY	**AVERAGE INSTRUCTION LENGTH (NON – ICALU)**	**AVERAGE INSTRUCTION LENGTH (WITH ICALU)**
Non–interlocked	$I_{PAVE1} = I_0 + 3.5$	$I_{ICAVE1} = I_0 + 4.17$
Interlocked	$I_{PAVE1} = 2I_0$	$I_{ICAVE2} = I_0 + 2.63$

Table 5.1 : Average Instruction Lengths for machines with and without ICALU

Using the values in the table, the total execution time for each machine can be calculated, for a given number of instructions.

1) COMPARISON OF TOTAL EXECUTION TIME :

The total execution time of a parallel machine is given as :

$$T_{NI} N_{NI} + T_I N_I \qquad (5.1)$$

Where,

T_{NI} = Time taken to execute a Non-Interlocked pair.

N_{NI} = Number of Non-Interlocked pairs.

T_I = Time taken to execute an Interlocked pair.

N_I = Number of Interlocked pairs.

Further,

$N = 2 (N_{NI} + N_I)$

$N_{NI} = ((100 – X) / 100) N / 2$, and

$N_I = (X / 100) N / 2$.

Where,

N = Total number of instructions to be executed.

X = Percentage of interlocked pairs.

Now, (5.1) can be rewritten as :

$$T_{NI} [((100 – X) / 100) N / 2] + T_I [(X / 100) N / 2] \qquad (5.1a)$$

Now, consider the following for a program :

a) N = 100,
b) X = 50 %
c) I_0 = 25 Logic Delays, typically

The execution times for the machines are :

I) NON-ICALU MACHINE :

From Table 5.1 :

$T_{NI} = I_{PAVE1} = I_0 + 3.5$.

$T_I = I_{PAVE2} = 2I_0$.

Substituting in (5.1a), we get,

$T_1 = (I_0 + 3.5) 25 + (2I_0) 25$

$= 1962.5$ Logic Delays.

II) MACHINE WITH ICALU :

Again from Table 5.1 :

$T_{NI} = I_{ICAVE1} = I_0 + 4.17$.

$T_I = I_{ICAVE2} = I_0 + 2.63$.

Substituting in (5.1a), we get,

Total execution time for 50 pairs of instructions,

$T_2 = (I_0 + 4.17) 25 + (I_0 + 2.63) 25$

$= 1419.78$ Logic Delays.

The machine with ICALU takes fewer logic delays than the Non-ICALU machine.

Chart 5.1 is a plot of (5.1a) with N constant (100) and varying X between 0 and 100 percent. It can be seen that the performance of the Non-ICALU machine degrades, where as the performance of the machine with ICALU is almost constant as X increases. This is because the Non-ICALU has to execute more and more instructions in serial. In the next section Percentage Speed Ratio is calculated.

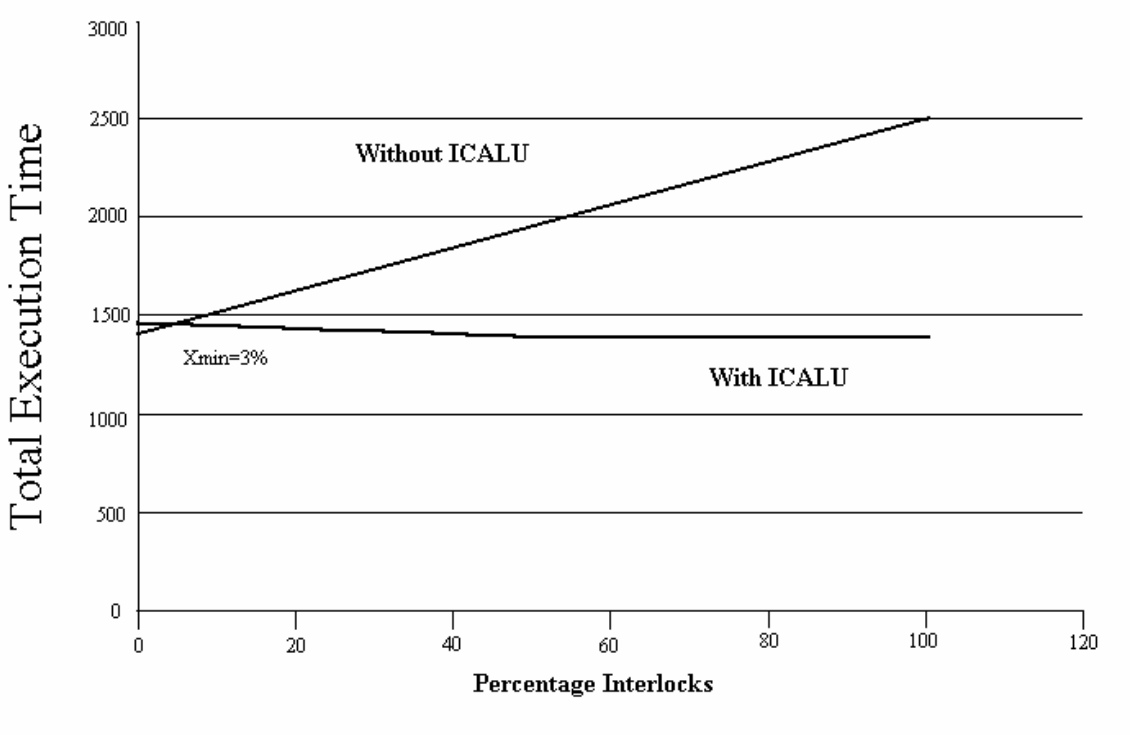


Fig 5.1 Percentage Interlocks Vs Total Execution Time

2) PERCENTAGE SPEED RATIO :

Percentage Speed Ratio of Machine 2 over Machine 1 is defined as :

$$[(T_1 - T_2) / T_1] \times 100 \qquad (5.2)$$

Percentage Speed Ratio reflects the time saved by one machine over the other.

Using (5.1a) in (5.2), we get,

$$[(T_{NI1} - T_{NI2}) (100 - X) + (T_{I1} - T_{I2}) X] / [T_{NI1} (100 - X) + T_{I1} X] \qquad (5.2a)$$

Hence,

Percentage Speed Ratio of machine with ICALU over the Non-ICALU machine for the previous case (that is X = 50%) ≈ 28

Similarly, for (say) X = 75% :
Percentage Speed Ratio ≈ 37.

Thus the Percentage Speed Ratio increases as X increases.

Chart 5.2 shows variation of Percentage Speed Ratio with interlock percentage (X). It can be seen clearly how Percentage Speed Ratio increases as interlock percentage (X) increases.

From chart we can see that at X ≈ 3%, the gain of the machine with ICALU is zero. Below this point the gain is negative, that is the machine with ICALU is slower than the machine Non-ICALU machine. This point can also be obtained by setting Percentage Speed Ratio to zero in (10.2a).

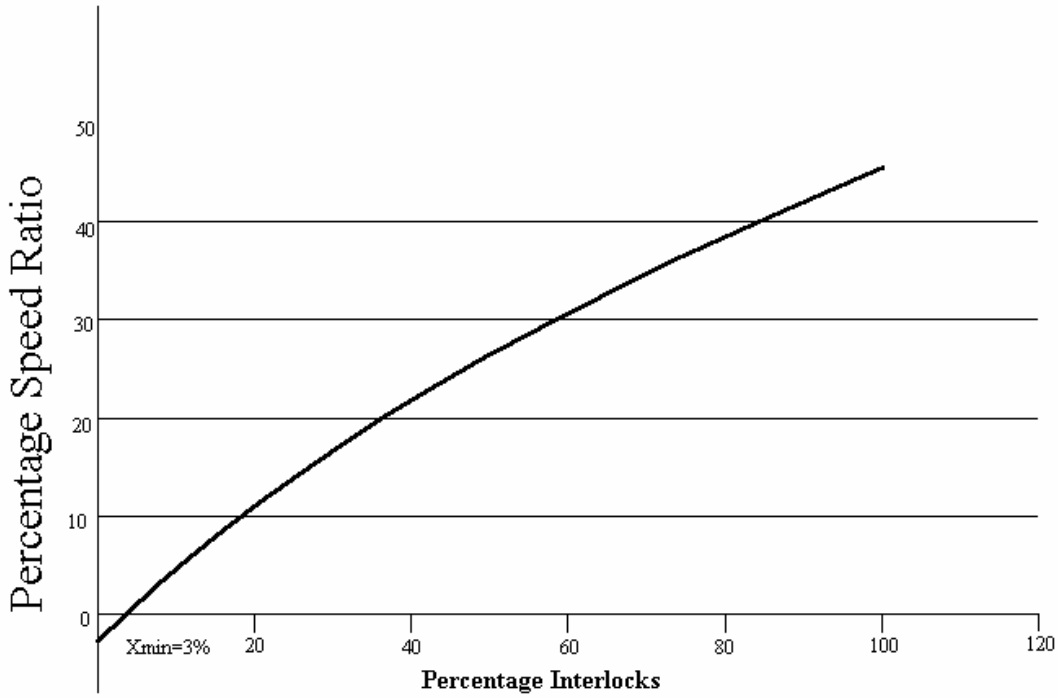


Fig : 5.2

(Percentage Interlock Vs. Percentage Speed Ratio.)

Chapter 6

TESTING PROCEDURES

TOTAL LOGIC AT A GLANCE:

INPUTS OF CSA

$K_1 \rightarrow 1$		$\rightarrow 0$		$\rightarrow 0$		$\rightarrow 0$			
$K_2 \rightarrow 1$	**Category 1**	$\rightarrow 0$	**Category 2**	$\rightarrow 1$	**Category 3**	$\rightarrow 0$	**Category 3**		
	(A ± B ± C)		**(A LOP B)± C**		**(A ± B) LOP C**		**(A LOP B)LOP C**		
$K_3 \rightarrow 0$		$\rightarrow 1$		$\rightarrow 0$		$\rightarrow 0$			

INPUTS OF PRE-CLA LOGIC BLOCK

$K_4 \rightarrow 0$		$\rightarrow 1$		$\rightarrow 1$		$\rightarrow 0$		$\rightarrow 0$	
$K_5 \rightarrow 1$	**AND**	$\rightarrow 0$	**NAND**	$\rightarrow 1$	**OR**	$\rightarrow 0$	**NOR**	$\rightarrow 1$	**XOR-XNOR**
$K_6 \rightarrow 0$		$\rightarrow 0$		$\rightarrow 0$		$\rightarrow 1$		$\rightarrow 1$	

INPUTS OF CLA

$K_7 \rightarrow 1$ M_1 Selects the $\rightarrow S_i$ (output of CSA) $K_7 \rightarrow 0$, M_1 selects $\rightarrow L_i$ (O/P of Pre-CLA Logic block)

ALU OUTPUT

$K_8 \rightarrow 1$ Arithmatic output $0 \rightarrow$ Logic (Mux 3)

INPUT OF POST CLA LOGIC BLOCK:

$K_9 \rightarrow 0$		$\rightarrow 1$		$\rightarrow 0$		$\rightarrow 0$			
	Arithmatic		AND,		OR,		XOR		
$K_{10} \rightarrow 0$	operation	$\rightarrow 0$	NAND	$\rightarrow 1$	NOR	$\rightarrow 0$	XNOR		
	of ICLUA								
$K_{11} \rightarrow 0$		$\rightarrow 0$		$\rightarrow 0$		$\rightarrow 1$			

ALU -1 O/P

$K_{12} \rightarrow 1$	O/P of	Arithmatic $\rightarrow 0$	Logic Operation	
$K_{13} \rightarrow 0$	ALU $\Rightarrow$	operation $\rightarrow 1$		

O/P OF ICALU

$K_{14} \rightarrow 0 \Rightarrow$ Non Inverted logical operation & Arithmatic operation(AND,OR,XOR)

$\rightarrow 1 \Rightarrow$ Inverted logical operation (NAND , NOR, XNOR)

TESTING

The ICUNIT has two outputs, result of ALU1 and that of ALU2. The testing of the ICUNIT was done by categories. They are as follows :

1) CATEGORY 1 (ARITHMETIC FOLLOWED BY ARITHMETIC) :

Since there are three operands, the four sub categories are :

i) All positive numbers.

ii) Two positive numbers.

iii) One positive number.

iv) None positive.

2) CATEGORY 2 (LOGICAL FOLLOWED BY ARITHMETIC) :

The sub categories are :

i) Logical AND followed by Arithmetic.

ii) Logical OR followed by Arithmetic.

iii) Logical XOR followed by Arithmetic.

iv) Logical NAND followed by Arithmetic.

v) Logical NOR followed by Arithmetic.

vi) Logical XNOR followed by Arithmetic.

3) CATEGORY 3 (ARITHMETIC FOLLOWED BY LOGICAL) :

The sub categories are :

i) Arithmetic followed by Logical AND.
ii) Arithmetic followed by Logical OR.
iii) Arithmetic followed by Logical XOR.
iv) Arithmetic followed by Logical NAND.
v) Arithmetic followed by Logical NOR.
vi) Arithmetic followed by Logical XNOR.

4) Category 4 (Logical followed by Logical) :

Category 2 and 3 cover all possible categories here. Hence only one subcategory is considered (say) :
Logical AND followed by Logical AND.

Chapter 7

SIMULATION RESULTS

SIMULATION RESULTS:

The simulation is conducted by assigning values to the variables in the design entities. The simulation is done through Modelsim XE II/starter 5.6e-Custom Xilinx Version. In Active-HDL a test run (simulation cycle) lasts for 100ns. The waveforms (resulting from the simulation) are displayed in waveform editor. The following pages show the simulation cycle as displayed by waveform editor.

The figures shown in the following pages depict the results of various categories of interlocked instructions explained in Chapter 6. A, B and C represents the three inputs to the ICUNIT. K1, K2, K3,…, K14 represents the different control signals. The figures show consecutive simulation cycles. Their values are shown in hexadecimal in each cycle.

OALU, OICALU are the outputs of ALU1 and ICALU respectively. In this OALU performs operation on A and B, whereas OICALU performs operation on the three operands.

ARITHMETIC FOLLOWED BY ARITHMETIC OPERATIONS

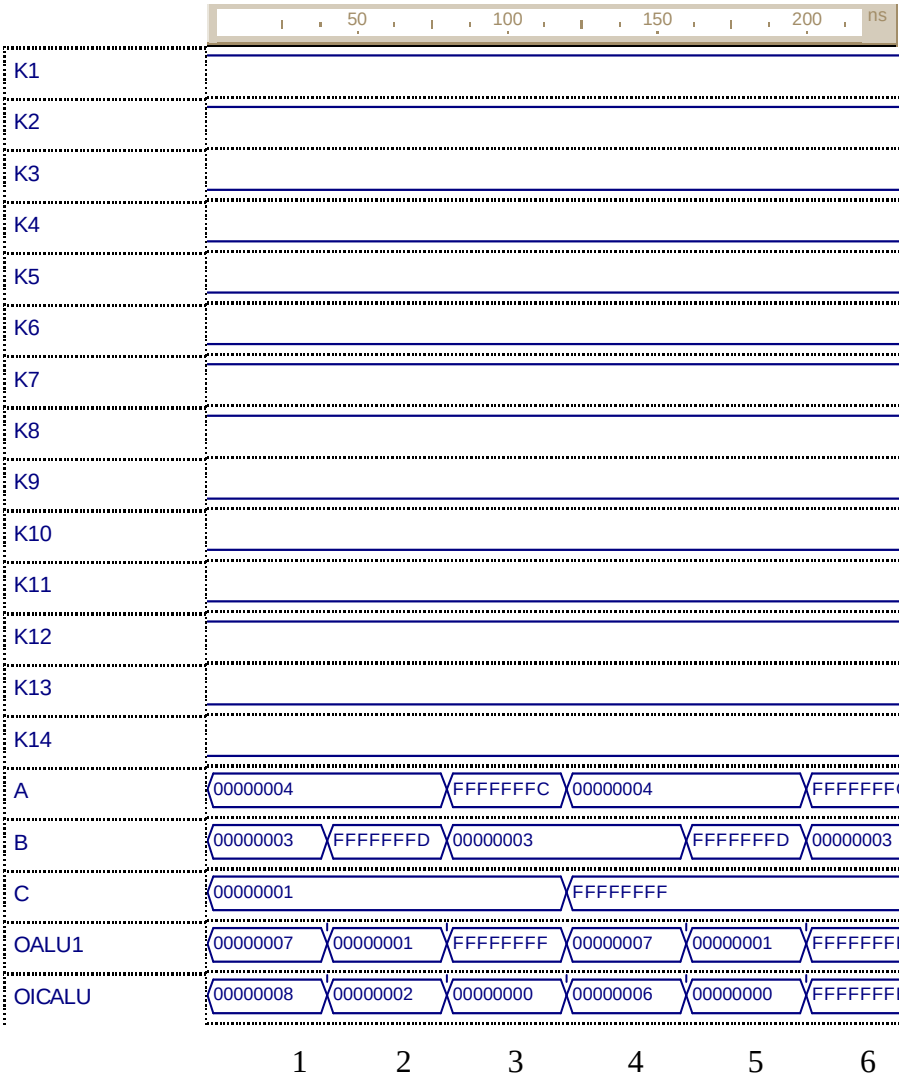


Fig 7.1

1. A + B + C
2. A – B + C
3. –A + B + C
4. A + B – C
5. A – B – C

6. –A + B – C

ARITHMETIC FOLLOWED BY LOGICAL OPERATIONS

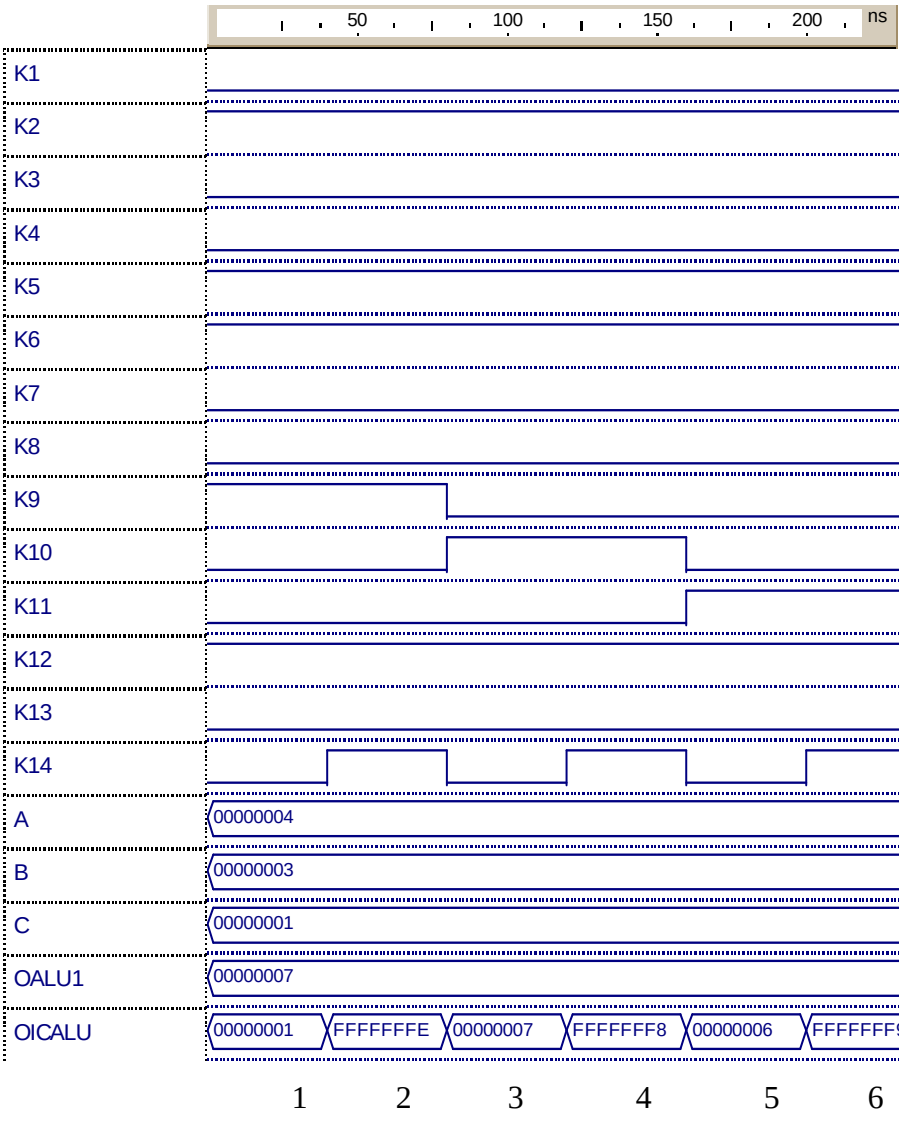
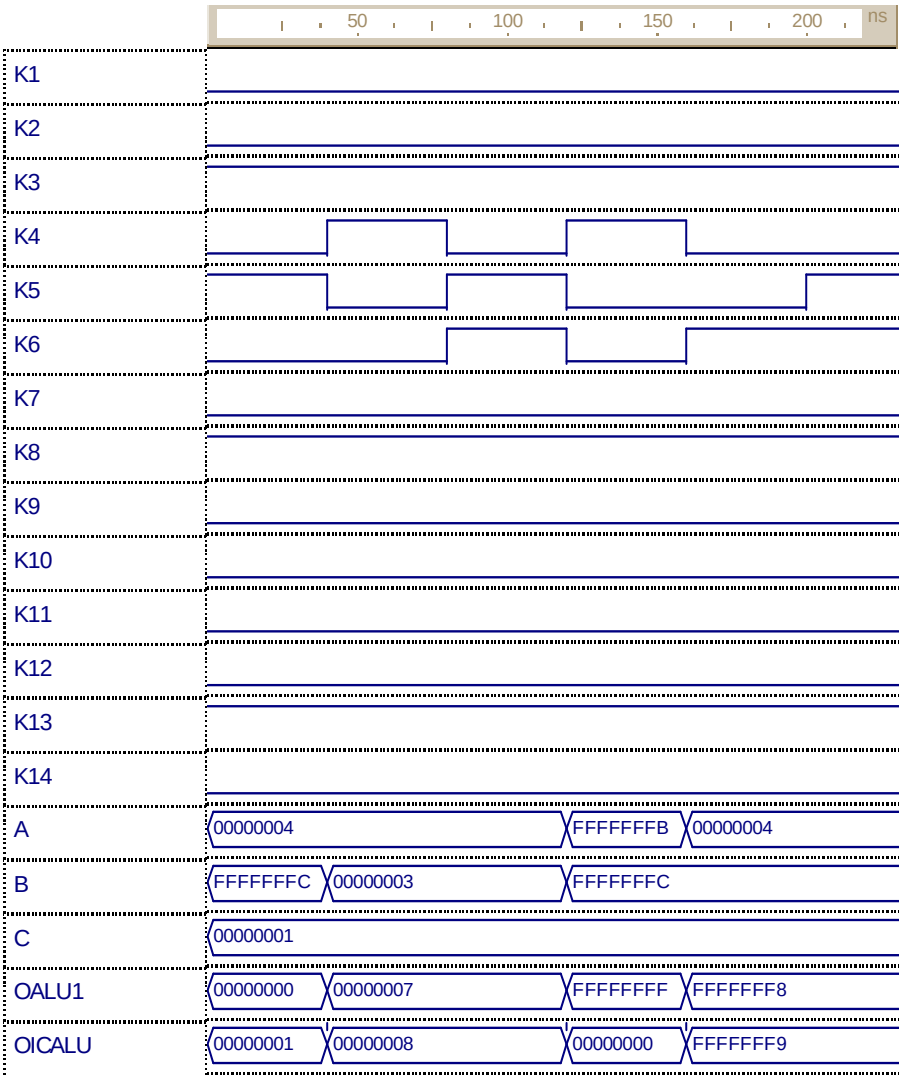


Fig 7.2

1. A + B **and** C
2. A + B **nand** C
3. A + B **or** C
4. A + B **nor** C
5. A + B **xor** C
6. A + B **xnor** C

LOGICAL FOLLOWED BY ARITHMETIC OPERATIONS



1 2 3 4 5 6

Fig 7.3

1. A **and** B + C
2. A **or** B + C
3. A **xor** B + C
4. A **nand** B + C
5. A **nor** B + C
6. A **xnor** B + C

LOGICAL FOLLOWED BY LOGICAL OPERATIONS

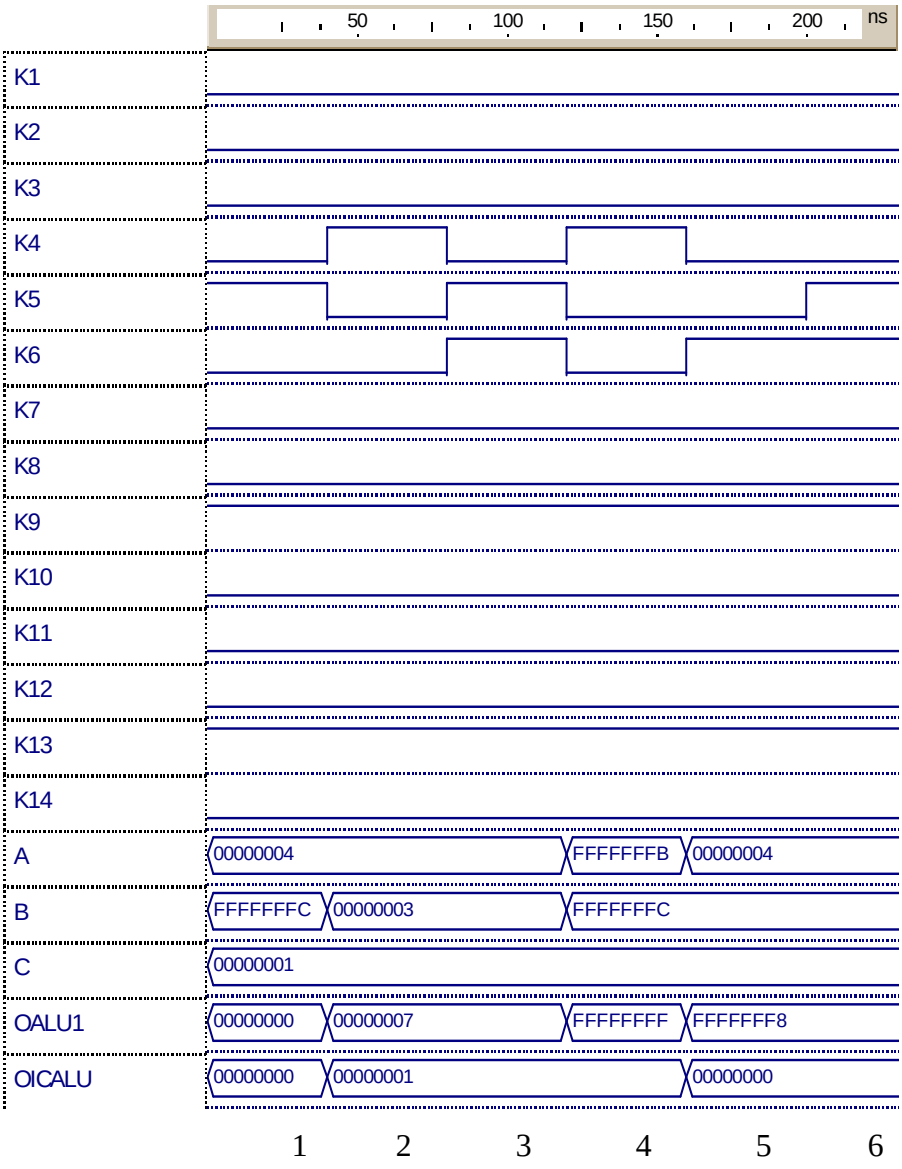


Fig 7.4

1. A **and** B **and** C
2. A **or** B **and** C
3. A **xor** B **and** C
4. A **nand** B **and** C
5. A **nor** B **and** C
6. A **xnor** B **and** C

Chapter 8

CONCLUSION

CONCLUSION

The objective of the thesis, execution of interlocked instructions in one instruction cycle. This was achieved by ICALU successfully designed and implemented using VHDL. Its functionality was verified through simulation.

The ICALU can be implemented in just 2 logic delays more than that of a conventional 2-1 ALU. The performance of an ordinary (Non-ICALU) parallel machine and the machine with the ICALU incorporated in it, was compared.

The following is concluded from the performance analysis :

- The Percentage Speed Ratio of the machine with the ICALU over the Non-ICALU machine depends only on the amount of interlocked instructions in the code and not on the total number of instructions.
- The Percentage Speed Ratio increases as the number of interlocked instructions increase. This is due to the degradation in performance of Non-ICALU machines.
- Assuming an average of (50-75)% interlocks in a given code, the Percentage Speed Ratio obtained is between (23-37)%, which implies that the ICALU, when incorporated in a parallel machine saves up to a third of the total execution time of the Non-ICALU machine.

Chapter 9

REFERENCE

REFERENCE:

1) J. Phillips, S. Vassiliadis, "High-Performance 3-1 Interlock Collapsing ALU's," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 257-268, Mar., 1994

2) D. W. Ruck, S. K. Rogers, M. Kabrinsky, M. E. Oxley, and B. W. Sutter, "The multilayer perceptron as an approximation to a Bayes optimal discriminant function,"IEEE Trans. Neural Networks, vol. 1, no. 4, pp. 296-298, Dec. 1990.

3) S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock collapsing ALU's,"IEEE Trans. Comput., vol. 42, no. 7, pp. 825-839, July 1992.

4) H. Ling, "High speed binary adder,"IBM J. Res. Develop., vol. 25, no. 3, pp. 156-166, May 1981.

5) M. J. Flynn and S. Waser,Introduction to Arithmetic for Digital Systems Designers. CBS College Publishing, 1982, pp. 215-222.

6) R. M. Keller, "Lookahead Processors," ***Computing Surveys,***Vol. 7, No. 4, pp. 514-537, December 1973.

7) R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," IBM J. Res. Develop., pp. 25-33, Jan. 1967.

8) R D Acosta , J Kjelstrup , H C Torng, An instruction issuing approach to enhancing performance in multiple functinal unit processors, IEEE Transactions on Computers, v.35 n.9, p.815-828, Sept. 1986

9) JAIN R.P . Digital Electronics , Printice hall

10) The Low Carb VHDL Tutorial ,Bryan Mealy 2004

SOURCE CODE

A.1 PROGRAM for ICUNIT

```
entity ICUNIT is
        port ( A, B, C : in BIT_VECTOR ( 31 downto 0 ) ;
        K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K12, K13, K14 : in BIT ;
        OALU1 : out BIT_VECTOR ( 31 downto 0 ) ;
        OICALU : out BIT_VECTOR ( 31 downto 0 ) ) ;
end ICUNIT ;

architecture B_ICUNIT of ICUNIT is

component ALU1
        port ( OP1, OP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        CNT1, CNT2, CNT3, CNT4, CNT5 : in BIT ;
        O : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

component ICALU
        port ( A, B, C : in BIT_VECTOR ( 31 downto 0 ) ;
        K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11, K14 : in BIT ;
        O : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

begin

        -- INSTATIATING ALU1

        CMP1 : ALU1 port map ( OP1( 31 downto 0 ) => A ( 31 downto 0 ),
        OP2 ( 31 downto 0 ) => B ( 31 downto 0 ),
        CNT1 => K4, CNT2 => K5, CNT3 => K6, CNT4 => K12, CNT5 => K13,
        O ( 31 downto 0 ) => OALU1 ( 31 downto 0 ) ) ;

        -- INSTATIATING ICALU
```

```
        CMP2 : ICALU port map ( A( 31 downto 0 ) => A ( 31 downto 0 ),
        B ( 31 downto 0 ) => B ( 31 downto 0 ),
        C ( 31 downto 0 ) => C ( 31 downto 0 ),
        K1 => K1, K2 => K2, K3 => K3, K4 => K4, K5 => K5, K6 => K6,
        K7 => K7, K8 => K8, K9 => K9, K10 => K10, K11 => K11, K14 => K14,
        O ( 31 downto 0 ) => OICALU ( 31 downto 0 ) ) ;

end B_ICUNIT ;
```

A.2 PROGRAM for ICALU

```
entity ICALU is
        port ( A, B, C : in BIT_VECTOR ( 31 downto 0 ) ;
        K1, K2, K3, K4, K5, K6, K7, K8, K9, K10, K11,K14 : in BIT ;
        O : out BIT_VECTOR ( 31 downto 0 ) ) ;
end ICALU ;

architecture B_ICALU of ICALU is

component CSA3_2
        port ( A, B, Z : in BIT_VECTOR ( 31 downto 0 ) ;
        K1, K2, K3 : in BIT ;
        Z_0_OUT : out BIT ;
        S : out BIT_VECTOR ( 31 downto 0 ) ;
        LAMBDA : out BIT_VECTOR ( 31 downto 1 ) ) ;
end component ;

component PREBLK
        port ( LL, LR : in BIT_VECTOR ( 31 downto 0 ) ;
        CON1, CON2, CON3 : in BIT ;
        L : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

component MUX_2_32
        port ( INP1, INP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        CNTRL : in BIT ;
        Y : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

component CLA
        port ( OP1, OP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        S : out BIT_VECTOR ( 31 downto 0 ) ) ;
```

```
end component ;



component P_CLA_LOGBLK
        port ( S, B : in BIT_VECTOR ( 31 downto 0 ) ;
        FADD, FAND, FORR, FXOR, FINV : in BIT ;
        O : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

signal S, L, INP1, INP2, SF : BIT_VECTOR ( 31 downto 0 ) ;
begin

        -- INSTANTIATING CSA
        CMP1 : CSA3_2 port map ( A( 31 downto 0 ) => A( 31 downto 0 ),
        B( 31 downto 0 ) => B( 31 downto 0 ),
        Z( 31 downto 0 ) => C( 31 downto 0 ),
        K1 => K1, K2 => K2, K3 => K3,
        S( 31 downto 0 ) => S( 31 downto 0 ),
        LAMBDA( 31 downto 1 ) => INP2( 31 downto 1 ),
        Z_0_OUT => INP2(0) ) ;

        -- INSTANTIATING PRE-CLA LOGIC BLOCK
        CMP2 : PREBLK port map ( LL( 31 downto 0 ) => A( 31 downto 0 ),
        LR( 31 downto 0 ) => B( 31 downto 0 ),
        CON1 => K4, CON2 => K5, CON3 => K6,
        L( 31 downto 0 ) => L( 31 downto 0 ) ) ;

        -- INSTATIATING MULTIPLEXER M1
        CMP3 : MUX_2_32 port map ( INP1( 31 downto 0 )
        => S( 31 downto 0 ),
        INP2( 31 downto 0 ) => L( 31 downto 0 ),
        CNTRL => K7,
        Y( 31 downto 0 ) => INP1( 31 downto 0 ) );

        -- INSTANTIATING CLASTAGE
        CMP4 : CLA port map ( OP1( 31 downto 0 ) => INP1 ( 31 downto 0 ),
        OP2( 31 downto 0 ) => INP2( 31 downto 0 ),
        S( 31 downto 0 ) => SF( 31 downto 0 ) ) ;

        -- INSTANTIATING POST CLA LOGIC BLOCK
        CMP5 : P_CLA_LOGBLK port map ( S( 31 downto 0 )
        => SF( 31 downto 0 ),B( 31 downto 0 ) => C( 31 downto 0 ),
```

```
        FADD => K8, FAND => K9, FORR => K10, FXOR => K11,
         FINV => K14 , O( 31 downto 0 ) => O( 31 downto 0 ) ) ;
          end B_ICALU ;
```

A.3 PROGRAM for ALU1

```
entity ALU1 is
        port ( OP1, OP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        CNT1, CNT2, CNT3, CNT4, CNT5 : in BIT ;
        O : out BIT_VECTOR ( 31 downto 0 ) ) ;
end ALU1 ;

architecture B_ALU1 of ALU1 is

component CLA
        port ( OP1, OP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        S : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

component PREBLK
        port ( LL, LR : in BIT_VECTOR ( 31 downto 0 ) ;
        CON1, CON2, CON3 : in BIT ;
        L : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

component MUX_22_32
        port ( INP1, INP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        CNTRL1, CNTRL2 : in BIT ;
        Y : out BIT_VECTOR ( 31 downto 0 ) ) ;
end component ;

signal SUM, LOG : BIT_VECTOR ( 31 downto 0 ) ;

begin

        -- INSTATIATING CLA
        CMP1 : CLA port map ( OP1( 31 downto 0 ) => OP1( 31 downto 0 ),
        OP2( 31 downto 0 ) => OP2(31 downto 0 ),
        S( 31 downto 0 ) => SUM(31 downto 0 ) ) ;

        -- INSTANTIATING LOGIC BLOCK
        CMP2 : PREBLK port map ( LL( 31 downto 0 )
```

```
                                        => OP1( 31 downto 0),
        LR( 31 downto 0 ) => OP2( 31 downto 0 ),
        CON1 => CNT1, CON2 => CNT2, CON3 => CNT3,
        L( 31 downto 0 ) => LOG( 31 downto 0 ) ) ;
        -- INSTATIATING MULTIPLEXER M1
        CMP3 : MUX_22_32 port map ( INP1( 31 downto 0 )
                                        => SUM(31 downto 0 ),
        INP2( 31 downto 0 ) => LOG( 31 downto 0 ),
        CNTRL1 => CNT4,
        CNTRL2 => CNT5,
        Y( 31 downto 0 ) => O( 31 downto 0 ) ) ;

end B_ALU1 ;
```

A.4 PROGRAM for 32-BIT CLA

```
entity CLA is
        port ( OP1, OP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        S : out BIT_VECTOR ( 31 downto 0 ) ) ;
end CLA ;

architecture B_CLA of CLA is

component CLA_1
        port ( X, Y : in BIT_VECTOR ( 7 downto 0 ) ;
        S : out BIT_VECTOR ( 7 downto 0 ) ;
        CIN : in BIT ;
        GRPGEN : out BIT ) ;
end component ;

component CLA_2
        port ( X, Y : in BIT_VECTOR ( 7 downto 0 ) ;
        CIN : in BIT ;
        S : out BIT_VECTOR ( 7 downto 0 ) ;
        GRPGEN, GRPPRP : out BIT ) ;
end component ;

signal C8, C16, C24 : BIT ;
signal INT1, INT2, INT3, INT4, INT5, INT6, INT7 : BIT ;

begin
```

```
    -- INSTANTIATING CLA_1
    C1 : CLA_1 port map ( X( 7 downto 0 ) => OP1 ( 7 downto 0 ),
    Y ( 7 downto 0 ) => OP2 ( 7 downto 0 ), CIN => '0',
    S ( 7 downto 0 ) => S ( 7 downto 0 ),
    GRPGEN => C8 ) ;


-- INSTANTIATING CLA_2
    C2 : CLA_2 port map ( X ( 7 downto 0 ) => OP1 ( 15 downto 8 ),
    Y ( 7 downto 0 ) => OP2 ( 15 downto 8 ),
    CIN => C8,
    S ( 7 downto 0 ) => S ( 15 downto 8 ),
    GRPGEN => INT2,
    GRPPRP => INT1 ) ;

    INT3 <= INT1 and C8 ;
    C16 <= INT2 or INT3 ;

    -- INSTANTIATING CLA_2
    C3 : CLA_2 port map ( X ( 7 downto 0 ) => OP1 ( 23 downto 16 ),
    Y ( 7 downto 0 ) => OP2 ( 23 downto 16 ),
    CIN => C16,
    S ( 7 downto 0 ) => S ( 23 downto 16 ),
    GRPGEN => INT5,
    GRPPRP => INT4 ) ;

    INT6 <= INT4 and INT2 ;
    INT7 <= INT4 and INT1 and C8 ;
    C24 <= INT5 or INT6 or INT7 ;

    -- INSTANTIATING CLA_2
    C4 : CLA_2 port map ( X ( 7 downto 0 ) => OP1 ( 31 downto 24 ),
    Y ( 7 downto 0 ) => OP2 ( 31 downto 24 ),
    CIN => C24, S ( 7 downto 0 ) => S ( 31 downto 24 ),
    GRPGEN => OPEN, GRPPRP => OPEN ) ;

end B_CLA ;
```

A.5 PROGRAM for 8 bit CLU

```
entity CLA_2 is
    port ( X, Y : in BIT_VECTOR ( 7 downto 0 ) ;
```

```
        CIN : in BIT ;
        S : out BIT_VECTOR ( 7 downto 0 ) ;
        GRPGEN, GRPPRP : out BIT ) ;
end CLA_2 ;
architecture B_CLA_2 of CLA_2 is

component SUM3_1
        port ( OP1, OP2 : in BIT ;
        OP3 : in BIT  ;
        Y : out BIT ) ;
end component ;

signal G0, P0, G1, P1, G2, P2, G3, P3, G4, P4, G5, P5, G6, P6, G7, P7 : BIT ;
signal C1, C2, C3, C4, C5, C6, C7 : BIT ;

begin

        -- GENERATION OF GENERATE AND PROPAGATE SIGNALS

        G0 <= X(0) and Y(0) ;       P0 <= X(0) or Y(0) ;
        G1 <= X(1) and Y(1) ;       P1 <= X(1) or Y(1) ;
        G2 <= X(2) and Y(2) ;       P2 <= X(2) or Y(2) ;
        G3 <= X(3) and Y(3) ;       P3 <= X(3) or Y(3) ;
        G4 <= X(4) and Y(4) ;       P4 <= X(4) or Y(4) ;
        G5 <= X(5) and Y(5) ;       P5 <= X(5) or Y(5) ;
        G6 <= X(6) and Y(6) ;       P6 <= X(6) or Y(6) ;
        G7 <= X(7) and Y(7) ;       P7 <= X(7) or Y(7) ;

        -- CARRY BLOCK 1
        SC1 : SUM3_1 port map ( X(0), Y(0), CIN, S(0) ) ;
        C1 <= G0 or ( P0 and CIN ) ;

        -- CARRY BLOCK 2
        SC2 : SUM3_1 port map ( X(1), Y(1), C1, S(1) ) ;
        C2 <= G1 or ( P1 and G0 ) or ( P1 and P0 and CIN ) ;
        -- CARRY BLOCK 3
        SC3 : SUM3_1 port map ( X(2), Y(2), C2, S(2) ) ;
        C3 <= G2 or ( P2 and G1 ) or ( P2 and P1 and G0 ) or ( P2 and P1 and
        P0 and CIN ) ;

        -- CARRY BLOCK 4
        SC4 : SUM3_1 port map ( X(3), Y(3), C3, S(3) ) ;
```

```
        C4 <= G3 or ( P3 and G2 ) or ( P3 and P2 and G1 ) or
    ( P3 and P2  and P1 and G0 ) or ( P3 and P2 and P1 and P0 and CIN ) ;


        -- CARRY BLOCK 5
        SC5 : SUM3_1 port map ( X(4), Y(4), C4, S(4) ) ;
        C5 <= G4 or ( P4 and G3 ) or ( P4 and P3 and G2 ) or
        ( P4 and P3 and P2 and G1 ) or ( P4 and P3 and P2 and P1 and G0 ) or
        ( P4 and P3 and P2 and P1 and P0 adn CIN ) ;

        -- CARRY BLOCK 6
        SC6 : SUM3_1 port map ( X(5), Y(5), C5, S(5) ) ;
    C6 <= G5 or ( P5 and G4 ) or ( P5 and P4 and G3 ) or
    ( P5 and P4 and P3 and G2 ) or ( P5 and P4 and P3 and P2 and G1 ) or
    ( P5 and P4 and P3 and P2 and P1 and G0 )
        or ( P5 and P4 and P3 and P2 and P1 and P0 and CIN ) ;

        -- CARRY BLOCK 7
        SC7 : SUM3_1 port map ( X(6), Y(6), C6, S(6) ) ;
        C7 <= G6 or ( P6 and G5 ) or ( P6 and P5 and G4 ) or
        ( P6 and P5 and P4 and G3 ) or ( P6 and P5 and P4 and P3 and G2 ) or
        ( P6 and P5 and P4 and P3 and P2 and G1 )
        or ( P6 and P5 and P4 and P3 and P2 and P1 and G0 ) or ( P6 and P5
        and P4 and P3 and P2 and P1 and P0 and CIN ) ;

        -- CARRY BLOCK 8
        SC8 : SUM3_1 port map ( X(7), Y(7), C7, S(7) ) ;
        GRPGEN <= G7 or ( P7 and G6 ) or ( P7 and P6 and G5 ) or ( P7 and P6
        and P5 and G4 ) or ( P7 and P6 and P5 and P4 and G3 ) or ( P7 and P6
        and P5 and P4 and P3 and G2 ) or ( P7 and P6 and P5 and P4 and P3
        and P2 and G1 ) or ( P7 and P6 and P5 and P4 and P3 and P2 and P1
        and G0 ) ;

        GRPPRP <= P7 and P6 and P5 and P4 and P3 and P2 and P1 and P0 ;

end B_CLA_2 ;
```

-- PROGRAM for CLA_1

For CLA_1 there will be no input carry and there will be no carry generation. Except from this the program remains the same as " CLA_2 ".

A.6 PROGRAM for CARRY SAVE ADDER

A.6.1

```
        entity CSA3_2 is
        port ( A, B, Z : in BIT_VECTOR ( 31 downto 0 ) ;
        K1, K2, K3 : in BIT ;
        Z_0_OUT : out BIT ;
        S : out BIT_VECTOR ( 31 downto 0 ) ;
        LAMBDA : out BIT_VECTOR ( 31 downto 1 ) ) ;
end CSA3_2 ;

architecture B_CSA3_2 of CSA3_2 is

component SUM3_1
        PROT ( OP1, OP2, OP3 : in BIT ;
        Y : out BIT ) ;
end component ;

component CSA_CARY
        PROT ( OP1, OP2, OP3, OP3_NXT, K1, K2, K3 : in BIT ;
        Y : out BIT ) ;
end component  ;

begin
        Z_0_OUT <= Z(0) and K3 ;
        SUM :
        for I IN 31 downto 0 generate
                SX : SUM3_1 port map ( A(I), B(I), Z(I), S(I) ) ;
        end generate SUM ;
        CARRY :
        for I IN 30 downto 0 generate
```

```
            CX : CSA_CARY port map ( A(I), B(I), Z(I), Z(I+1), K1, K2, K3,
            LAMBDA(I+1) ) ;
        end generate CARRY ;

        end B_CSA3_2 ;
```

A.6.2

```
entity SUM3_1 is
        port ( OP1, OP2, OP3 : in BIT ;
        Y : out BIT ) ;
end SUM3_1 ;

architecture B_SUM3_1 of SUM3_1 is

begin
        Y <= ( OP1 and not OP2 and not OP3 ) or
        ( not OP1 and OP2 and not OP3 ) or
        ( not OP1 and not OP2 and OP3 ) or
        ( OP1 and OP2 and OP3 ) ;
end B_SUM3_1 ;
```

A.6.3

```
entity CSA_CARY is
        port ( OP1, OP2, OP3, OP3_NXT, K1, K2, K3 : in BIT ;
        Y : out BIT ) ;
end  CSA_CARY ;

architecture B_CSA_CARY of CSA_CARY is
begin
        Y <= ( K2 and OP1 and OP2 ) or
        ( K1 and OP2 and OP3 ) or
        ( K1 and OP1 and OP3 ) or
        ( K3 and OP3_NXT ) ;
end B_CSA_CARY ;
```

A.7 PROGRAM for PRE-CLA LOGIC BLOCK

A.7.1

```
entity PREBLK is
        port ( LL, LR : in BIT_VECTOR ( 31 downto 0 ) ;
```

```
        CON1, CON2, CON3 : in BIT ;
        L : out BIT_VECTOR ( 31 downto 0 ) ) ;
end PREBLK ;


architecture B_PREBLK of PREBLK is

component PRELOG
        port ( OP1, OP2, P1, P2, P3 : in BIT ;
        Y : out BIT ) ;
end component ;

begin

        ITERATE :
        for I IN 31 downto 0 generate
             PX : PRELOG port map ( LL(I), LR(I), CON1, CON2, CON3, L(I) ) ;
        end generate ;
end B_PREBLK ;
```

A.7.2

```
entity PRELOG is
        port ( OP1, OP2, P1, P2, P3 : in BIT ;
        Y : out BIT ) ;
end PRELOG ;

architecture B_PRELOG of PRELOG is

begin
        Y <= ( OP1 and P1 ) or ( OP2 and P1 ) or
        ( OP1 and NOT OP2 and P2 ) or ( not OP1 and OP2 and P3 ) ;
end B_PRELOG ;
```

A.8 PROGRAM for POST-CLA LOGIC BLOCK

A.8.1

```
entity P_CLA_LOGBLK is
        port ( S, B : in BIT_VECTOR ( 31 downto 0 ) ;
```

```
        FADD, FAND, FORR, FXOR, FINV : in BIT ;
        O : out BIT_VECTOR ( 31 downto 0 ) ) ;
end P_CLA_LOGBLK ;



architecture B_P_CLA_LOGBLK of P_CLA_LOGBLK is

component P_CLA_BCMP
        port ( X, B, FADD, FAND, FORR, FXOR, FINV : in BIT ;
        Y : out BIT ) ;
end component ;

begin

        OUTPUT_STAGE :
        for I IN 31 downto 0 generate
                PX : P_CLA_BCMP port map ( S(I), B(I), FADD, FAND, FORR,
                FXOR, FINV, O(I) ) ;
        end generate ;
end B_P_CLA_LOGBLK ;
```

A.8.2

```
entity P_CLA_BCMP is
        port ( X, B, FADD, FAND, FORR, FXOR, FINV : in BIT ;
        Y : out BIT ) ;
end P_CLA_BCMP ;

architecture B_P_CLA_BCMP of P_CLA_BCMP is

signal INT1, INT2, INT3 : BIT ;

begin

        INT1 <= FADD or ( B and FAND and not FINV ) or
        ( FORR and not FINV ) or ( not B and FXOR and not FINV ) or
        ( B and FXOR and FINV ) ;

        INT2 <= ( B and FXOR and not FINV ) or ( FAND and FINV ) or
```

```
        ( not B and FORR and FINV ) or ( not B and FXOR and FINV ) ;

        INT3 <= ( B and FORR and not FINV ) or (not B and FAND and FINV) ;

        Y <= ( X and INT1 ) or ( not X and INT2 ) or INT3 ;

end B_P_CLA_BCMP ;
```

A.9

```
entity MUX_2_32 is
        port ( INP1, INP2 : in BIT_VECTOR ( 31 downto 0 ) ;
        CNTRL : in BIT ;
        Y : out BIT_VECTOR ( 31 downto 0 ) ) ;
end MUX_2_32 ;

architecture B_MUX_2_32 of MUX_2_32 is

component MUX2
        port ( OP1, OP2, P1 : in BIT ;
        Y : out BIT ) ;
end component ;

begin

        ITERATE :
        for I IN 31 downto 0 generate
                PX : MUX2 port map ( INP1(I), INP2(I), CNTRL, Y(I) ) ;
        end generate ITERATE ;
end B_MUX_2_32 ;

entity MUX1 is
        port ( OP1, OP2, P1, P2 : in BIT ;
        Y : out BIT ) ;
end MUX1 ;

architecture B_MUX1 of MUX1 is

begin
        Y <= ( OP1 and P1 ) or ( OP2 and P2 )  ;
end B_MUX1 ;
```

DETERMINATIN OF INSTRUCTION LENGTHS FOR FREQUENTLY EXECUTED INSTRUCTIONS

Since the instructions to a machine are of two types, Interlocked and Non-interlocked, Average instruction lengths are calculated separately for both categories for each machine.

1) Interlocked Instructions :

For this category the Non-ICALU parallel machine always executes in serial. Hence,

Average Instruction Length of Non-ICALU machine for interlocked category =

$T_{PAVE2} = 2\ I_0$ L.D. (L.D. = Logic or gate Delay)

For the machine with ICALU, instructions can be further classified as having :

i) Same Destination registers :

Here both the registers have same destination registers. The possibilities are :

a) R_{D1}, R_{S1}

R_{D1}, R_{S2}

Time required = (I_0 + 2) L.D.

b) R_{D1}, R_{S1}

R_{D1}, R_{S1}

Time required = (I_0 + 2) L.D.

c) R_{D1}, R_{D1}

R_{D1}, R_{D1}

Time required = (I_0 + 2) L.D.

d) R_{D1}, R_{D1}

R_{D1}, R_{S2}

Time required = (I_0 + 3) L.D.

ii) Different destination registers :

e) R_{D1}, R_{S1}

R_{D2}, R_{D1}

Time required = (I_0 + 3) L.D.

f) R_{D1}, R_{S1}

R_{D2}, R_{D1}

Time required = (I_0 + 3) L.D.

g) R_{D1}, R_{S1}

R_{S1}, R_{D1}

Time required = (I_0 + 3) L.D.

h) R_{D1}, R_{D1}

R_{S2}, R_{D1}

Time required = (I_0 + 3) L.D.

The Average instruction length of the machine with ICALU for the interlocked category is :

$T_{ICAVE2} = [\ 3(I_0 + 2) + 5(I_0 + 2)\ ] / 8 = (I_0 + 2.63)$ L.D.

2) Non-Interlocked Instructions :

In the non-interlocked category R_{D1} can never be present in the second instruction. Hence the sub-classification for Interlocked category does not apply here. Two classifications are possible here, which are :

i) One common register between two instructions :

a) R_{D1}, R_{D1}
R_{S2}, R_{D1}
Time required for Non-ICALU machine = (I_0 + 3) L.D.
Time required for machine with ICALU = (I_0 + 3) L.D.

b) R_{D1}, R_{D1}
R_{S2}, R_{D1}
Time required for Non-ICALU machine = (I_0 + 3) L.D.
Time required for machine with ICALU = (I_0 + 3) L.D.

ii) No common registers between two instructions :

c) R_{D1}, R_{D1}
R_{S2}, R_{D1}
Time required for Non-ICALU machine = (I_0 + 3) L.D.
Time required for machine with ICALU = (I_0 + 5) L.D.

d) R_{D1}, R_{D1}
R_{S2}, R_{D1}

Time required for Non-ICALU machine = (I_0 + 6) L.D.

Time required for machine with ICALU = (I_0 + 8) L.D.

e) R_{D1}, R_{D1}

R_{S2}, R_{D1}

Time required for Non-ICALU machine = (I_0 + 3) L.D.

Time required for machine with ICALU = (I_0 + 3) L.D.

f) R_{D1}, R_{D1}

R_{S2}, R_{D1}

Time required for Non-ICALU machine = (I_0 + 3) L.D.

Time required for machine with ICALU = (I_0 + 3) L.D.

Average instruction length for Non-ICALU machine with non-interlocked instructions =

T_{PAVE1} = [5(I_0 + 3) + (I_0 + 6)] / 6 = (I_0 + 3.5) L.D.

Average instruction length for ICALU machine with non-interlocked instructions =

T_{ICAVE1} = (I_0 + 4.17) L.D.